Behavior Tree Library

Bachelor's Thesis

Video Game Design and Development Degree

Travila Cuadrado, Guillem

2018

Pillosu González, Ricard

Acknowledgments

I would like to thank Ricard Pillosu for his expert advice and dedication throughout this project. You took time out of where it does not exist to help me.

This project would have been impossible without the support of StackOverflow who has been always there to solve my doubts.

Summary

Behavior tree[1] popularity has grown in video game AI, however there is not any library in C/C++ that is not engine-dependant, has a visual editor and a visual debugger. Through a set of weekly milestones and with the help of tools like: GitHub and Trello, we built a library that meets the initial objectives and it is ready to be extended in the developer's desired direction.

Keywords

video game, artificial intelligence, behavior tree, library.

Index

Tables index

Figures index

Glossary

AI - Artificial Intelligence: in video games this term refers to the action of mimic cognitive functions that humans associate with other human minds, such as learning and problem solving

FSM - Finite state machine: mathematical model of computation used in video games to represent entity behaviors. It is formed by a set of states where the FSM can change from one state to another by some external inputs.

GDC - Game Developers Conference: primary forum where professionals of all roles involved in the video game development gather to exchange ideas and shape the future of the industry.

NPCs - Non-playable-characters: in a game, any character that is not controlled by the player, therefore by the computer.

UE4 - Unreal Engine 4: commercial 3D video game engine. Is one of the industry-leading engine's for its power, performance and trust from developers.

1. Introduction

1.1 Motivation

The main motivation to do this project is to create a tool that others can use in a professional or academic environment.
Throughout the degree, several tools given to the community for free have been beneficial in the process of learning. For this reason, the motivation of this project is to express gratitude to all these developers by contributing with a tool that speeds up and saves time at some point of the development process.

1.2 Problem formulation

Behavior trees[1] have become one of the most used and powerful tools in the video game industry in the artificial intelligence field. A well-constructed tool providing the key features to work with behavior trees in a fast and agile way would save a lot of time, resources and, in consequence, a lot of money.

Big studios and commercial engines already have their own adaptation to work with behavior trees. However, the number of free libraries that provide the same functionality is very reduced. Due to this, integrating artificial intelligence in a self made engine or project results a very time consuming task.

The video game industry has the need of free, open-source libraries to allow developers to create their own technology without the necessity to make it all from scratch. Time would be better invested if the main focus was creating that specific new technology instead of implementing systems that have already been created. This would result in an enormous advance of the technology used in video games.

Having described the current situation of the industry, the problem that we will focus on will be to fulfill this gap of free, open-source tools or libraries available to speed up the development process. Our target field will be artificial intelligence in video games.

There are already several libraries that provide a useful range of tools to create a robust AI. Some of the core modules in AI are already solved. For instance, pathfinding already have Recast, the state of the art navigation mesh construction toolset for games. However, the

core module in AI development, the decision making, does not have a solid well-constructed toolset.

As it has been said at the beginning, the industry is tending toward behavior trees when designing the decision making of their AI. For all of the reasons mentioned above, the problem we aim to focus is to provide a toolset to work with behavior trees that can be integrated in any project without the need to use a specific commercial game engine.

1.3 General objectives

The general objectives of this project is to provide a tool that has:

- Practical use in both, academic and professional, contexts
- Simple integration
- Handle all the decision making logic
- Visual and clear method to create and edit behavior trees
- Make accessible the debug process

1.3.1 Practical use

The final objective of this project is to have a tool that can actually be used in the real world. Showcase this project as another one in the portfolio without having a practical use is a situation that needs to be avoided.

We aim to focus to two distinct targets: academic and professional. For academic or learning purposes, it should be simple enough to start right away and see practical results as soon as possible. The reason to try to achieve this objective is to contribute to the learning of behavior trees. On the opposite side, it should also be complex enough to satisfy the majority requirements of a professional project.

1.3.2 Simple integration

In order to fulfill the previous objective, it should also have a simple integration to start working within a few minutes.

One of the main reasons developers choose not to use a library is a complex integration. The first step when working with a new library is the integration. If the effort to integrate the

library is too high, probably the time saved when using it is spent on the first step. This is the cause why, at that moment, the highest percentage of people quit. To avoid it, an easy integration provides the opportunity to test the library in a short time and makes it more accessible to new users.

1.3.3 Handle the decision making logic

The final objective of using a behavior tree is to let it handle the decision making. With this tool we provide a way to create behavior trees as well as executing them without the need to implement all the internal logic.

For this reason, one of the main objectives is to manage the logic that comes with behavior trees themselves. The final user will respond to the information that this tool brings about the state of the tree without concerning about the internal side of it.

1.3.4 Visual edition of behavior trees

Behavior trees are not only a great tool to deal with decision making, they are also an excellent method to visualize and understand an entity's behavior. If the creation of trees is done through code, the comprehension of the behavior turns more complex and obscure.

In order to keep this advantage of behavior trees, it is crucial to have a visual representation of the tree throughout its edition.

1.3.5 Accessible debug process

Debugging is a fundamental part of every software developing process. Having the right tools when debugging can enormously increase the production performance. This is the reason why an accessible debug process can be profitable when working with behavior trees.

When talking about accessible, we are referring to the capability to easily understand what is happening in order to find out what is not working as expected as soon as possible. The debug mode must show the information to the user in a way that is easy to understand and at the same time complex enough to have access to all the information available.

1.4 Specific objectives

Taking into account the general objectives, the specific objectives are designed to concrete even more the result expected to have a clear vision of what we aim to accomplish.

Each specific objective inherits from a general objective and makes a detailed explanation of the goal that is planned to be reached. For this reason we have divided the specific objectives into different paragraphs grouped by the general objective that have in common.

1.4.1 Practical use

This general objective was about keeping the simplicity of the tool to make it accessible for new adopters while maintaining the versatility of a powerful tool that can be used in a professional environment.

The specific objectives encapsulated in this bigger one are:

- Having a cost of adoption smaller than five minutes
- Keep the code ready to scale it in any direction without any limitation
- Minimal impact in performance

In the first one, when we use the term ‘cost of adoption’ we are referring to the minimum time that the user will need to spend to learn the basics of our tool and be able to have a practical result. The five minutes period is small enough to keep the user’s attention while discovers the possibilities of the tool and large enough to have a valuable and meaningful result. With this specific objective we want to cover the goal of having a tool simple to use.

The second specific objective is about having the base classes generalized enough to give the option to build upon it. What is trying to be avoided with this objective is to create some sort of limitation from the beginning that precludes the user to use our library because it is not possible to scale it into the user’s desired direction.

Finally, the last specific objective is set to ensure the tool is ready for a professional environment. Our library is a subsystem of a bigger one, AI, inside the whole game and for this reason the performance consumed by this should not be higher than half millisecond in a modern computer. These benchmarks are set for a regular use case, we are supposing that we need to update around twenty behavior trees every tick.

1.4.2 Simple integration

Integrate a library can be a tedious work and, if the process is not clear, some may end up moving to another library. For all the reasons explained in the general objective, the time of integration should not be too long and the steps of the process should be as clear and concise as possible to avoid confusing the user.

In this case, we made two specific objectives to reach our goal:

- The time of integration is around 10-30 minutes.
- A well-documented guide is provided explaining step by step the process in the simplest and clearest manner.

1.4.3 Handle the decision making logic

In this part we want to hide all the internal logic of the execution process of a behavior tree from the user. It is our system that takes care of this and what is shown to the user is only the final result of this process.

As a consequence, the specific objective is:

- Provide function calls to initialize, run and finalize behavior trees that handles all their internal process by themselves as well as offering additional methods to check the state of the tree and query relevant information.

1.4.4 Visual edition of behavior trees

Visual edition will be another differentiative feature of our tool. The specific objectives proposed for this one are:

- Visualize the tree the same way a user would have in a schema or diagram.
- Present all properties that could be modified through code in a user-friendly interface

1.4.5 Accessible debug process

This general objective is related to the previous one because both share the same purpose, avoid working with behavior trees through code and offer the maximum range of options through a graphical user interface.

Then, the debug process will share the same specific objectives as the visual editor:

- Visualize the state of the tree as it is shown in the editor while the user is debugging
- Present all properties, during the debug mode, in a user interface to be able to track all the steps the tree did while it was running.

1.5 Scope of the project

This project intents to reach a wide range of audience. As it has been said before, this product is for people who want to learn or experiment with behavior trees and also the ones who plan to use it professionally.

Knowing that there is a fixed time limitation to develop it, we will reduce the size of the project to the minimal valuable product. With this, we mean that what we will develop will be a solid base to extend it in the desired direction of each user. For instance, if someone is planning to use it professionally, he/she will be able to scale it to the needs of its project.

It is complicated to reach a wide target as we have defined, taking in consideration the limitations at the time to make this project, although if we plan to create a general base extensible in multiple directions the objectives proposed and the scope of the project become more realistic and reachable.

2. State of the art

2.1 History

Behavior trees have emerged from the necessity to create better AIs and to keep up with the continuous expanding and demanding video game industry. Before them, the decision-making was handled by finite state machines (FSM). They provided a very simplistic and clear way to represent and also visualize the behavior of an entity. However, this model became unsustainable as the behaviors of the entities grew in size and complexity.

FSM are still used in video game AI to handle simple and small behaviors. It continues to be the best option to represent a very limited behavior that is not expected to grow. Nevertheless, FSM have some disadvantages that promoted the appearance of behavior trees, the main disadvantage is that does not work well with scalability. The number of

states grows very fast as soon as multiple behaviors are implemented and it loses one of the core advantages, the easy visualization of each state. For this reason, behavior trees were introduced to solve the problem of complexity at the same time that keep a clean representation.

The origin of behavior trees was in 2005 at GDC presented by Damian Isla in his conference “Proceeding: Handling Complexity in the Halo 2 AI”. It was oriented to model behaviors of NPCs. Since then, they have been used extensively in numerous video games such as Bioshock, Halo, Crysis and many more. Its use have also been adopted by other fields, mainly robotics. Because of this, behavior trees have been evolving through the years and are now at their maturity stage. To back the previous statement, there are some facts that prove their mature state:

- Game engines have already integrated BTs as one of their systems inside the AI module. Some examples are: Unreal Engine and CryEngine.
- Game AI books are already treating BTs as one of the techniques used in video game AI.

2.2 Behavior trees in game engines

Before starting to describe the state of the art there are some subsections that need to be said. At the current time of this project there are not any state of the art related to our topic inside the academia. It is not surprising considering that the video game industry is very recent and therefore there are not any consolidation of an academia yet. For this reason we will look for the state of the art in the market, what are big companies using and what developers tend to use while they have to create a product professionally.

There are a lot of game engines nowadays so we will focus only on the most used and that are available for any developer. The reason to do this is because the most used engines are the ones that provide the most advanced technology and are always evolving to keep up with the state of the art and, of course, with their competitors. In-house engines will not be discussed here because it is impossible to get information about their technology for obvious reasons.

Therefore we will discuss in more detail the implementation of behavior trees in the following engines:

- Unreal Engine 4
- Unity
- CryEngine

2.2.1 Unreal Engine 4

Unreal Engine 4 already has a behavior tree tool integrated inside their framework. This tool allows the user to create, edit and debug a behavior trees all inside a visual editor.

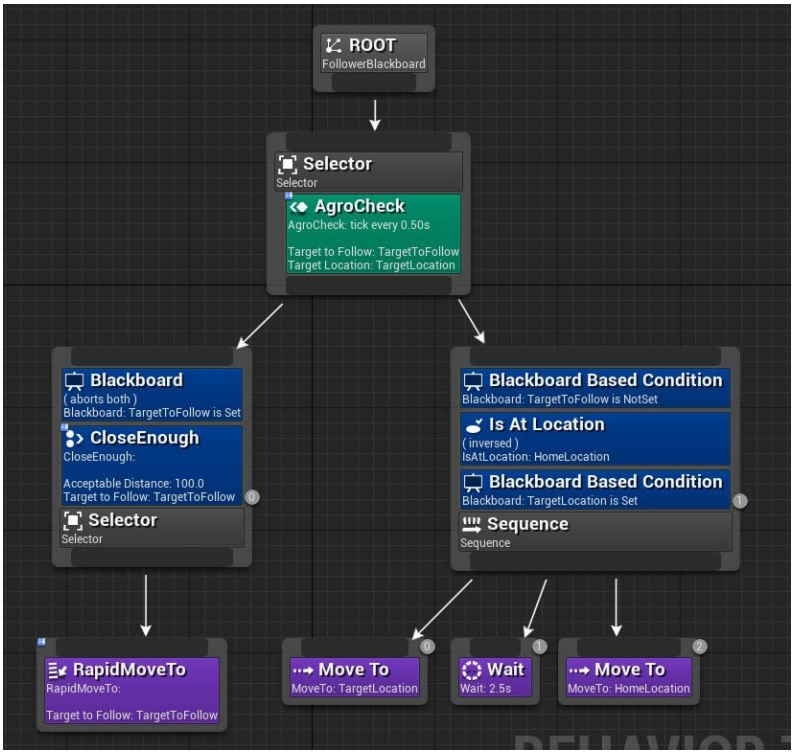


Figure 2.1 Unreal Engine 4 Behavior Tree Editor

The advantages of having a tool like this already integrated in your engine are:

- No need of any implementation to use behavior trees as resources
- Easy interconnection between components. Ex: the code can reference the tree since the engine has knowledge of both. The tree can use an object in the scene and access to all of its information as well as modify it.

- Fluid debug process as a result of a complete integration between the tool and the engine. Not having to jump from one tool to another while debugging increases the amount of information that can be shared in the process making it easily for the developer to use.

Unreal Engine states itself that there are three critical ways where their approach differs from the “standard” behavior tree model:

1. Behavior trees are event-driven
2. Conditionals are not leaf nodes
3. Special handling for concurrent behaviors

Behavior trees are event-driven

This approach avoids doing lots of work every frame because BTs passively listen for “events” which can trigger changes in the tree instead of constantly checking whether any relevant change has occurred.

Event-driven behavior trees are part of the named “second generation”[2] that aims to improve performance.

Conditionals are not leaf nodes

In the “traditional” BT model, conditionals are classified as tasks and therefore are leaf nodes. Making conditionals be decorators has a couple of advantages according to UE4:

- Make the behavior tree UI more intuitive and easy to read
- It is easy to make decorators act as observers (wait for events) at certain nodes in the tree and in consequence gaining full advantage from the event-driven nature of the trees.

Special Handling for concurrent behaviors

UE4 has removed parallel nodes and instead they offer two solutions to handle concurrent behaviors:

- Simple Parallel
- Services

Simple parallel nodes only allow two children: a single task node and the other of which can be a complete subtree.

Services are special nodes associated with any composite node which can register for callbacks every certain seconds and perform updates that need to occur periodically.

UE4's reason for this change is that their new approach works better with event-driven behavior trees and it can be easier to optimize rather than using the "standard" model.

2.2.2 Unity

Unity is one of the top choices for developers when picking an engine. At the moment of writing the current Unity's version is 'Unity 2017.3' and yet there are not any implementation of behavior trees integrated inside the engine.

However, Unity allows to create tools for the engine relatively easy and share it in their online store, the Asset Store. There are a few advantages of this:

- There is a variety of same tools to choose from. This means that a developer can pick the one that adjusts better to the project.
- Once the asset is purchased, there is completely freedom to edit or change the code and there is total transparency about how the tool is made.

On the other hand, it also brings some disadvantages:

- It may not be optimized enough or the performance would not be the same that if it was natively implemented in the engine.
- There is no guarantee that the tool will be updated: to fix bugs or to adapt to new versions of the engine.

There are two main assets related to behavior trees that need to be highlighted, and another one that is worth mentioning:

- Behavior Designer
- NodeCanvas
- NPBehave

Behavior Designer and NodeCanvas

Both packages offer the same functionality. There are little differences between them but the core features are present in both:

- Visual editor
- Visual Debugger
- Data-Oriented
- Built in event system
- Provided API to extend functionality on your own

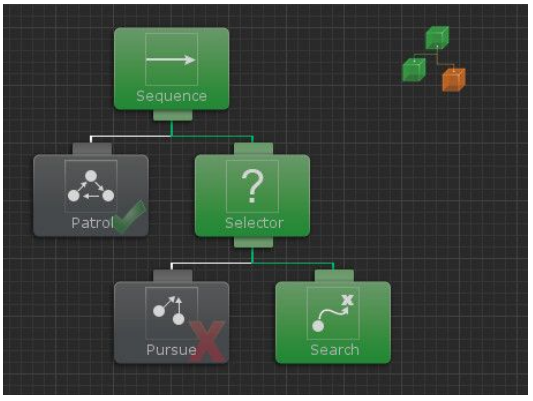


Figure 2.2 Behavior Designer editor

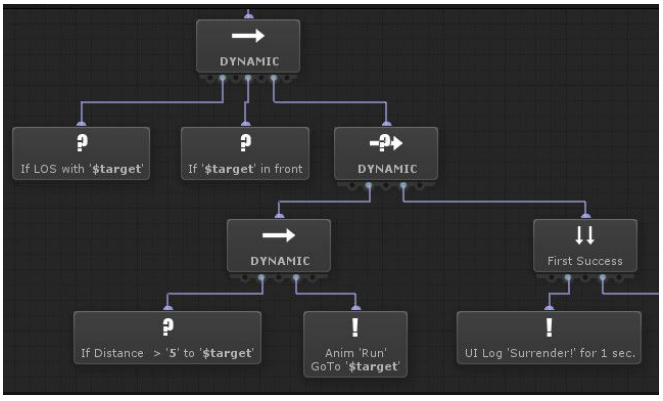


Figure 2.3 NodeCanvas editor

The main difference between these and the one integrated in UE4 is that in Unity both are Data-Oriented while in UE4 is Event-Driven. These two architectures are part of the second generation of behavior trees focused in performance and optimization.

NPBehave

NPBehave is an Event-Driven behavior tree package, unlike the previous. It is worth mentioning because, currently, it is the only one that is event-driven.

However, it cannot be compared with the other ones because of its simplicity. It lacks too many features to compete with the others (visual editor, API, proper documentation…). It is also relevant to highlight that this package free and the source code is available in GitHub for everyone who wants to contribute. This explains the difference between packages, while this one was designed to contribute to the community altruistically, the other two were focused on commercial business.

2.2.3 CryEngine

CryEngine has a unique system to work with behavior trees. The user needs to code an XML file and place it in the engine to read it. Continuously, the engine reads the XML file and creates a behavior tree that is ready to be used.

ExampleTree.xml

```
<BehaviorTree>
    <Variables>
        <Variable name="HasTarget"/>
    </Variables>
    <SignalVariables>
    </SignalVariables>
    <Timestamps>
    </Timestamps>
    <Root>
        <Sequence>
            <Log message="Test" />
            <WaitForEvent name="OnEnemySeen" />
            <Move to="Target" speed="Walk" stance="Stand" fireMode="BurstWhileMoving" />
            <Halt />
        </Sequence>
    </Root>
</BehaviorTree>
```

Figure 2.4 Example of a XML file of a behavior tree in CryEngine

For the debug process, CryEngine offers two different methods:

- Visual representation in-game printing the XML structure through text on to the screen

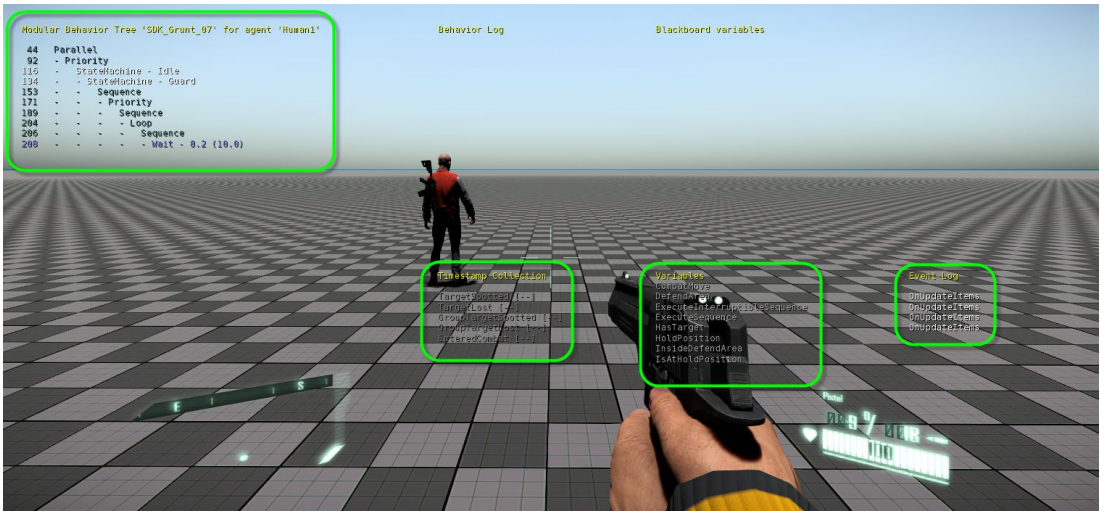


Figure 2.5 In-game debugging of a behavior tree in CryEngine

- Saving a log file with the same information as the previous method but with the advantage of keeping track of all the changes during the game test.

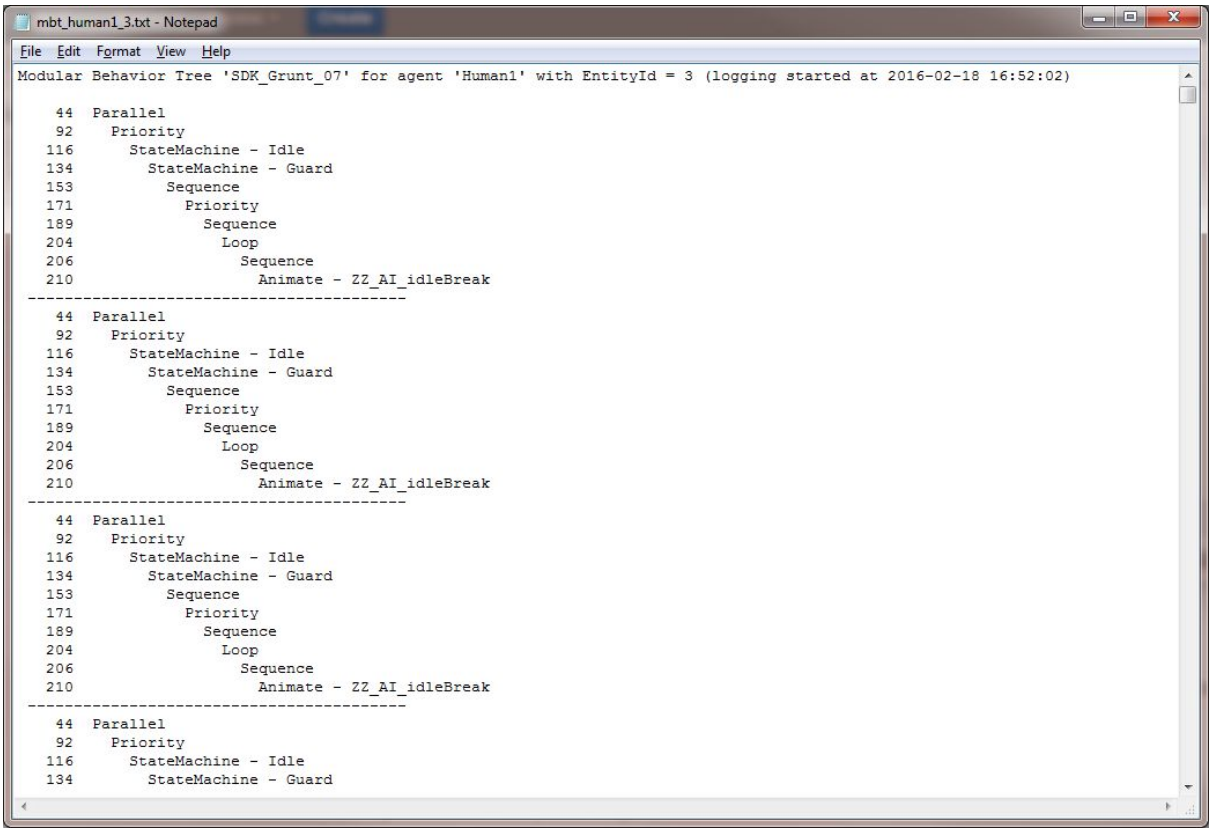


Figure 2.6 Log file of a behavior tree debug session in CryEngine

2.3 Behavior tree libraries

Opposite to the previous implementations there are some libraries that offer a way to create behavior trees and that can be implemented without being engine depending.

Using a library external to the engine has some advantages:

- No need to use a specific engine, you can work with your own.
- More control over the workflow between the engine and the library

However, it also brings some disadvantages:

- Needs an initial time cost of integration
- The synchronization of the library with the engine will not be ideal. Usually the libraries are generic to fit into a wide range of projects, because so, it will not be optimized with your engine and the performance will not be the best possible.

At the moment of writing this paper the behavior tree libraries that are available for free are (excluding the ones with poor functionality):

- Behavior3
- BrainTree
- BT++

2.3.1 Behavior3

A framework that provides a set of tools to create, design and use behavior trees. It features a visual editor to create behavior trees. The editor runs on a browser and, as a result, does not depend on other tools.

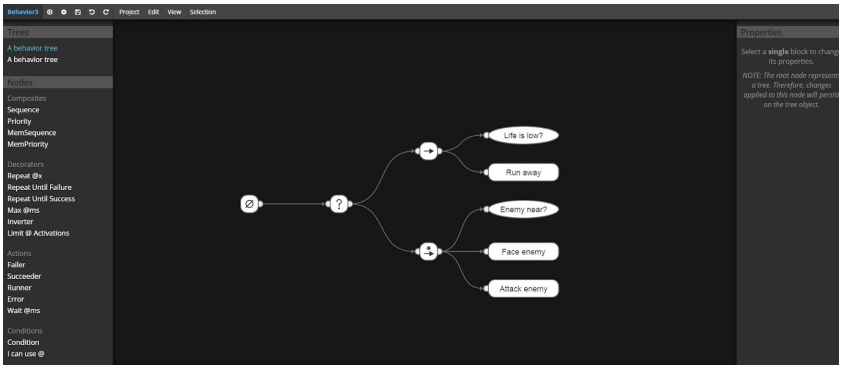


Figure 2.7 Behavior3 editor

Everything, the editor and the framework, are open-source to freely adapt it to everyone's needs. The editor serializes the tree in a JSON file to take it and load it to your engine.

It also provides two libraries two load and use behavior trees that are written in:

- JavaScript
- Python

2.3.2 BrainTree

A C++ behavior tree header-only library. The upsides are that is very simple and compact but the downsides are that is too much simple and does not provide lots of functionalities a part from the most basics (work with behavior trees, predefined composites, predefined decorators, a blackboard).

2.3.3 BT++

A behavior tree library in C++. It contains more functionality than the previous one as the framework allows to work with the nodes in a more extens way.

The code is open source and a manual is given to understand how it works and been able to expand its functionality. However, all the behavior trees need to be created through code and does not provide a serialization system neither.

2.4 State of the art: comparison and conclusion

Once that all the current solutions are described a general comparison is needed in order to quickly visualize the strengths and the weakness of each one:

Name	Dependency	Language	Visual editor	Visual debugger	BT generation	Features
UE4 BT	UE4	C++	✓	✓	2nd Event-Driven	**4 / 5**
Behavior Designer	Unity	C#	✓	✓	2nd Data-Oriented	**3 / 5**
NodeCanvas	Unity	C#	✓	✓	2nd Data-Oriented	**3 / 5**
NPBehave	Unity	C#	✗	✗	2nd Event-Driven	**1 / 5**
CryEngine	CryEngine	XML	✗	✓	Not specified	**1 / 5**
Behavior3	-	JS/Python	✓	✗	Not specified	**2 / 5**
BrainTree	-	C++	✗	✗	Not specified	**1 / 5**
BT++	-	C++	✗	✗	Not specified	**1 / 5**
Ideal solution	-	C++	✓	✓	2nd generation	**5 / 5**

Table 2.1 Behavior tree solutions comparison

Our ideal solution would be a library that does not rely on any engine in particular, written in C++, with a visual editor, a visual debugger and that works with the second generation of behavior trees.

The choice of not being engine dependent and be written in C++ is because our goal is to provide a tool that can be integrated in self-made engine and because C++ is the most used language in video games it has to be written in this language for a better implementation.

In conclusion, we have seen that the state of the art is slightly defined but still needs more time to consolidate. We have also observed that there is no similar tool in the market that provides all the functionality that we plan to offer all in one tool.

3. Planning

The project will be divided into the following stages:

- Pre-production: v.0.0
- Vertical Slice: v.0.1
- Alpha: v.0.2
- Beta: v.0.5
- Gold: v.1.0

Each stage will have a version number associated. Between stages different versions can appear in between. Ex: Alpha Stage, versions released during Alpha: v.0.2, v.0.21, v.0.245…

Once the stage is reached the official version will be released with the release tag, the other releases in between will have the pre-release tag.

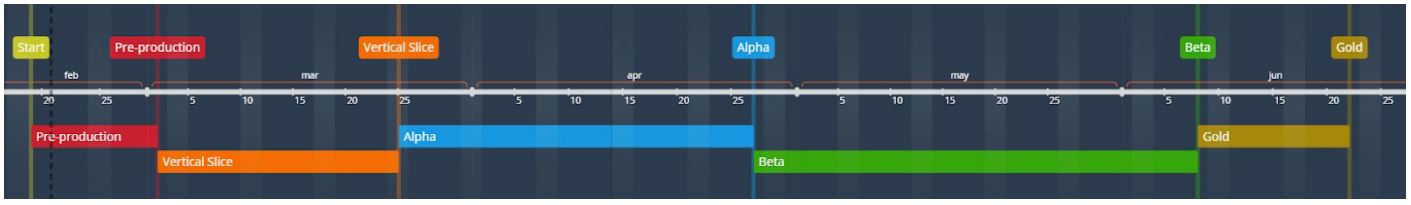


Figure 2.5 Project stages timeline

The timeline will be as following:

Stage	End date	Period
Start project	19 Feb	-
Pre-production	2 Mar	11 days
Vertical Slice	25 Mar	23 days
Alpha	27 Apr	33 days
Beta	8 Jun	42 days
Gold	22 Jun	14 days

Table 2.2 Project stages dates and periods

3.0 Project stages

3.0.1 Pre-production

In this stage the main concern is to investigate and test the tools that will be used and have not worked with. It is crucial to explore the possibilities that these new tools can offer to detect if something is missing or if it is not the right tool for this.

The other part of this stage is focused on setting up all the other tools that we have already worked with and will be used in this project. It is about preparation, having everything ready to use before starting the project itself.

Regarding the investigation, the tools/libraries to be investigated or that the expertise with it doesn't reach the minimums required are:

- Node Graph Editor: an extension library of ImGui to work with nodes
- Networking and SDL_net 2.0: concerning the network side, we will make use of the SDL_net 2.0 library, mainly because our application already uses SDL and it is a multiplatform library.

3.0.1.1 Node Graph Editor

Node Graph Editor is an implementation of a node editor with the ImGui API. The creator describes it as a basis for more complex solutions, this means that probably the library is already prepared to grow while maintaining consistency. It is available for Windows, macOs and Linux and only requires GLFW3 to be installed in the developer's system. Both, the platform and the library, match the requirements set up previously for the project. As a result, the use of this library will not need additional implementations in order to work.

The goal of this subpart of the preproduction is to learn to use it and find out if needs an additional implementation to fit the project requirements. Unfortunately, there is not a documentation of the API and there are not almost any comment in the code, making the process to understand it more difficult and slow.

3.0.1.2 Networking and SDL_net 2.0

Networking is another fundamental part of the project because it handles the communication of the library (integrated in the user's application) with the graphical application that we provide.

Taking in consideration that we program both sides, the control that we have over the communication process is absolute. Although this could be considered as a strength it is also a weakness because we have to make sure that the whole communication process is optimal, efficient and secure. Having this high level of responsibility together with the fact that our networking skill is not enough to reach the expectations is the reason why a networking research is indispensable.

For this subpart, the objective is to work with the sockets library that SDL provide, SDL_net 2.0, and practise until we master the connection and transfer of data between two applications. To simplify it, because mastering networking is a very long process and could take more time than we have, we will resize the learning content to be able to transfer data between to applications running on the same computer through a local network. These conditions will be the most common ones when a user is working with our tool, that is why we will target the most probable cases instead of trying to handle all the possible scenarios. However, there is always the possibility to expand our tool to satisfy specific situations but this is up to the user as we don't have the time to provide such a wide range of options.

3.0.2 Vertical Slice

The Vertical Slice will try to touch a little bit of every core system to forecast future conflicts and will act as a test to validate if our initial approach goes in the right direction. At the end of the stage there will be a playable prototype that will show the main features of the tool.

It is important to point out that this version will not contain all the functionality and the features implemented could or not work at all their extension. It is also expected to have issues unresolved without being a problem at that moment. As we mentioned before, the goal is to have an application running that can bring us an idea of the possibilities of this tool and what could become of it.

For the reasons mentioned above, the core features that this stage should have are (ordered by descending priority):

1. Visual node editor to create, link and delete nodes
2. Serialization system to save and load nodes in the visual editor
3. Loader to read the file created by the editor and creates a behavior tree structure
4. Library functions to init the system and close it
5. Library function to load a behavior tree generated with the editor

3.0.3 Alpha

In this stage everything should be running except the debugger. It should be possible to create behavior trees and use it in a project with our library implemented. There are room for some issues that can be solved in later stages.

The new content of this version will be:

1. Blackboard added in the editor
    a. Create, edit, link variables of the blackboard
    b. Serialize
    c. Load and use it in the library
2. The library loads and runs the behavior trees created in the editor with all its functionality, this means every node will act as it should act

3.0.4 Beta

This stage will be to finish all the features of the tool, basically the debugger, and close the project. After this stage no more features will be added. The goal of this stage is to have the product finished with all its functionality.

The new changes of this stage will be:

1. Connection between the library and the editor in runtime.
2. Visualize the state of the trees running through the library in the editor
3. Save the debug sessions to load it and recreate it later on step by step (only visually in the editor)
4. Provide the function calls to edit the debug options through the library before running the program

3.0.5 Gold

After this stage the product will be finished with all its functionality. During gold, the only two tasks that will be performed will be:

1. Bug fixing
2. Polish

There is no way to predict the issues that will come with this project, because of that the goal will be to fix as many bugs as possible, always starting for the most critical and leaving to the end the minor imperfections. Furthermore, the tool will need some polish to have a professional look and to present itself as a finished product.

3.1 Analysis SWOT, risks and contingency plan

The SWOT analysis of the project can be broken down into:

Strengths

- Free open-source: no entry barriers for developers to try and build upon it
- Visual representation: makes it accessible for non-programmers and easier for programmers
- Editor and library separated: allows to work independently the programmer and the designer

Weaknesses

- Solo developer: limited amount of resources to spend in the project
- No visibility: hard to reach the target the tool is destined for because there is not any platform to share it and gain visibility. Furthermore, the industry does not know us yet as we have not done anything relevant and neither have we entered to work in the industry

Opportunities

- Few competitors: there are any tool similar to what we aim to offer or the functionality that provides is much more inferior
- Behavior trees are extensible used nowadays in video game AI

Threads

- Chance of a dead market: if behavior trees are used that much and nobody has already developed a similar tool considering that behavior trees are in their mature stage, it may be possible that there is not a necessity for this kind of tool
- Very small market sector: this tool is only for custom software, not so many developers create their own technology because the vast majority tend to use a commercial engine.

In the following paragraphs we will define each risk and its contingency plan associated to cope with it:

- Risk - Unexpected issue appears making unable to follow with the project
    - Plan - Try to return to a previous version that does not contain this issue and continue from there. If not possible, pivot and aim the project to a viable direction
- Risk - Incapacity to execute a task due to a lack of knowledge.
    - Plan - Ask for help or directions to a more experienced person in the field. If not found, ask for someone that could provide help and contact him/her.
- Risk - Not having enough time to finish
    - Plan - Evaluate the project every stage and modify the objectives accordingly to the obtained results so far.

3.2 Initial costs analysis

The hypothetical cost of this project taking into account that the development time is around 5 months and with only one person working on it would be:

Costs	Subtype	Per unit	Units	Number of units	Amortization period (monthly)	Period used in the project	Total
Personel	Programmer	3,000 €	months	5	-	-	15,000 €
Goods and services	Internet	40 €	months	5	-	-	200 €
	Electricity	70 €	trimesters	1.6	-	-	112 €
Equipment	PC	800 €	items	1	36	5	111 €
	Monitor	100 €	items	1	36	5	14 €
	Keyboard	80 €	items	1	36	5	11 €
	Mouse	40 €	items	1	12	5	17 €
	Chair	100 €	items	1	24	5	21 €
	Desktop	70 €	items	1	60	5	6 €
Total costs							15,491 €

Table 2.3 Project costs

3.3 Management tools

Trello and GitHub will be the two tools used to manage this project.

On one hand, Trello will keep track of all the tasks pending, in progress, to test and completed through the different stages.

On the other hand, GitHub will take care of the management of all the bugs and issues that may appear.

4. Methodology

A set of milestones will divide the project into different deliveries. These milestones coincide with the different stages of the project described in the planification section.

Additionally, an agile methodology will be introduced to keep a closer track. The Scrum methodology is the one that fits better for our kind of project. The characteristics of our Scrum will be the following:

- Weekly sprints: from Monday to Sunday
- Sprint planning: every Monday
- Sprint review: every Sunday

There are also some aspects of this application that differ from the traditional Scrum methodology:

- No roles: it does not make sense in a one person project
- No daily stand ups: no need to communicate the daily progress if there is only one member

4.1 Tools for the monitoring of the project

Different tools will be used to keep track of the project, each one will be for a very specific purpose:

- Trello: to monitor the advances of the tasks
- GitHub: this tool offers multiple functionality and we will take advantage of only some of them:
    - GitHub issues: will keep a record of all the issues and the changes of the stage of each one
    - GitHub releases: will show the progress of the project in a executable form to test the product
- Git Bash: the version control selected. It will follow all the changes made in the code

4.2 Validation tools

Two different heuristics will be validated along the project: utility and quality.

- Utility: refers to the degree in which each version reaches the expected functionalities.
- Quality: the degree in which the version is stable enough and free from bugs or issues

The first one, utility, will be validated by the director of this project. He will decide, in his criteria, if the version offers the functionality predicted. His judgement will be considered the definition of done of each task.

The second one, quality, will be self evaluated with a series of testing rounds at each sprint review to identify all the issues and bugs at the current stage of the project. There is also the possibility to allow other users to test and try a version of our product in mid-development. If the case occurs, it will also serve as a validation tool to recognize possible problems that early adopters of the tool may have.

5. Development

The development process, as has been said before, is divided in the traditional stages the video game development process. In the following chapters we will discuss the performance in each stage according to our initial planification. At the beginning of each chapter there will be a reminder of the objectives proposed during the planification phase.

5.1 Pre-production

Objectives:
- Set up the tools need it for the project.
- Explore the possibilities of the new tools:
    - Node Graph Editor: ImGui's extension to create node graphs.
    - SDL_net 2.0: SDL's extension to handle network

The first step was to set up all the tools mentioned in the Methodology chapter. This included the creation of a:

- Trello board: to be filled with all the tasks for each stage
- GitHub repository: to have a version control of the project's code

Trello's board was configured to be reusable for each stage. The following lists were created to keep a track of each individual task status:

- **TO DO**: contains all the tasks of the current stage
- **IN PROGRESS**: tasks currently being developed
- **TO TEST:** tasks finished but need a revision to assure all functionality works.
- **DONE:** tasks finished, tested and without any issues.
- **Stage [delivery date]:** informative list to keep in mind the current stage and the finishing date. Example: *'Alpha [27 - Apr] '*
- **LIMBO:** new tasks that come up during the development that are: not crucial, enhancements or discarded due to time and irrelevancy.

The next step was to create the GitHub repository, a process well-known and repeated multiple times during the degree. We also created a README and a LICENSE file to finish setting up the repository.

The license chosen was the MIT license that:

- Allows: modification, distribution, private use, commercial use
- Does not have any warranty or liability of any kind. Making the author not responsible of any damage that the software could produce.
- Requires the copyright and license to be included in all copies or portions of the software.

These were the conditions that best suited our project. To summarize, it allows other developers to do anything with the code without any legal responsibility by the author but with the obligation to credit the author for the work.

Once the repository was set up, the next step was to create a minimum base of code to start exploring these new tools. To do that we needed a basic window to display all our tests in a visual and clear way.

Since we already worked in the past with SDL and OpenGL, we decided that it was the best option to quickly have a window where we could render our content. The reasons why we chose both libraries were:

- SDL:
  - Cross-platform
  - Easy access to: audio, keyboard, mouse and graphics
  - Written in C and works natively with C++
  - Easy to use
  - Well documented
  - Stable
- OpenGL:
  - Cross-platform
  - Usable with C and C++
  - Easy to use
  - Well documented
  - Stable

Due to the limited time to develop this project, third party libraries are required to ease those tasks that are not the main focus of research.

After all the tools were set up we moved to the investigation of the new tools. These were:

- Node Graph Editor
- SDL_net 2.0

5.1.1 Exploration: Node Graph Editor

Node Graph Editor is a library that relies on ImGui (graphical user interface library). Evidently, the first step was to implement ImGui, an easy task considering that we already implemented and used this library multiple times in the past.

With the core library implemented we started to implement the extension library. It didn't suppose a problem considering that it only needed a few calls to get it up and running. However, when we started to explore the possibilities of the library the process became difficult and slow.

The goal was to be able to create nodes and link them among them. That would be the main use of the library, anything extra would be a plus. With that goal in mind, we started to investigate the options of the library and we ended with the following conclusions:

1. It offers enough possibilities to reach the needs of the project
2. Its use will be slow and difficult due to:
    a. A lack of documentation in the GitHub page.
    b. A lack of comments in the code

Some code examples are provided with the Node Graph Editor library but it is still a tough task to understand the code and be able to use it. Because of this, the time provided for this task was not enough and it took a bit longer to complete.

5.1.2 Exploration: SDL_net 2.0

Same as with Node Graph Editor, SDL_net also relies on another library, SDL. However, we already implemented SDL when we were creating a basic window, which speeded up the integration of SDL_net.
In this case there was a modification of the initial planning. We considered that explorating the capabilities of the SDL_net at that point was irrelevant for two reasons:

1. There wasn't any code to test the functionality. This library was originally picked to handle the network connections between our editor and our library when debugging. At that point we only had one window, so in order to explore SDL_net we would have had to implement at least the root of the library and the editor.
2. We realised that it was more valuable to invest that time exploring in more depth the node editor library than the network library. Mainly because the network library will only be used for debugging, while the node editor will be an essential part of the project.

5.1.3 BeeT 0.01

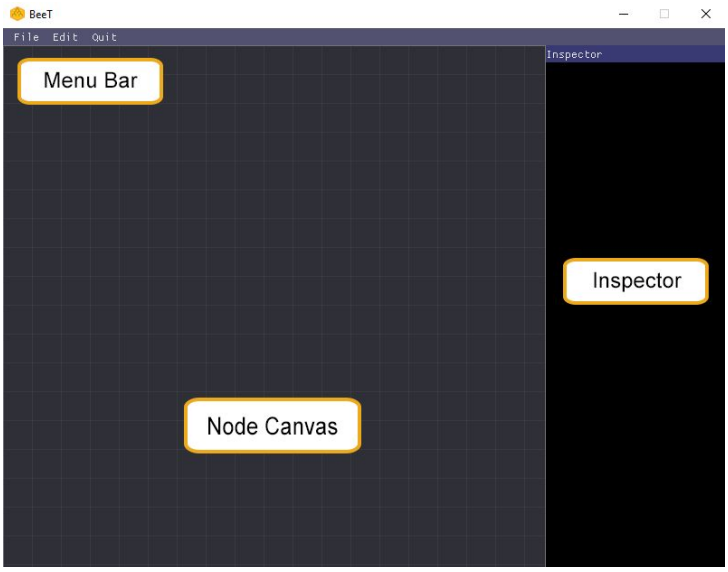
Following the planning, at the end of each stage there will be a deliverable with the specified version number.

In this case, the pre-production corresponds to the version 0.01. A build of the project was made and placed in the *Releases* section in the GitHub page. Each version made before the official release contains a description with the new changes compared to the previous one.

These changes are called *ChangeLog*. The resulting demo contains the following changes:

- GUI (editor) solution project
- SDL and OpenGL(Glew) libraries integrated. Both linked in static.
- SDL_net library integrated
- Main menu bar
- Node Canvas
- Inspector
- Creation of nodes by type
- Remove nodes
- Link nodes
- Remove links
- Save a Behavior Tree in a file
- Load the Behavior Tree file

The editor app looks like this:



Menu Bar: contains the basic application options.

Inspector: shows information about the selected node

Node Canvas: canvas where the node graph will be displayed

5.1 BeeT 0.01 editor

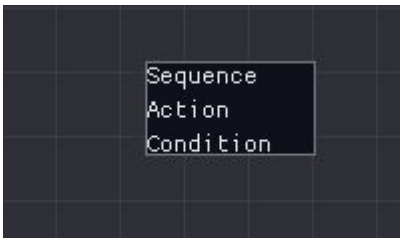
The *File* tab in the *Menu Bar* contains:

- The current directory path
- An option to *Open* a behavior tree file
- An option to Save the current behavior tree in a file

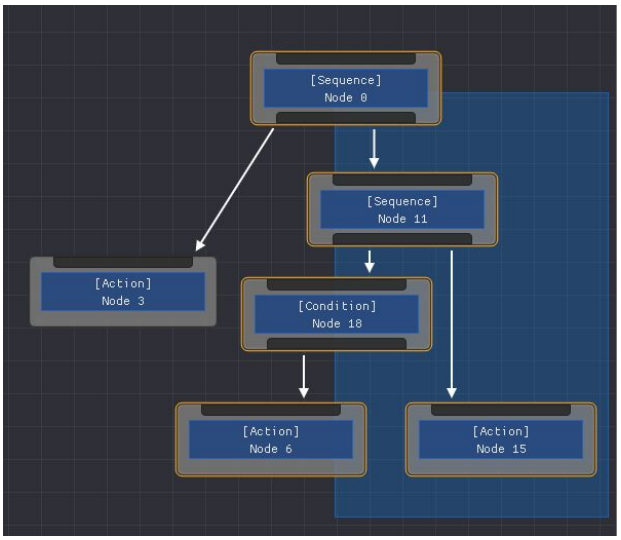
5.2 BeeT 0.01 File tab

At that point the behavior tree was automatically saved in the *current directory path* and loaded also from there. The user could not specify the save location neither the name of the resulting file yet. It only had the basic load/save functionality to test that it worked.

The creation of nodes is done with the right mouse button. A menu is displayed to select the node type. Selection is done either clicking with the left mouse button or dragging with the same button to multi-select several nodes.



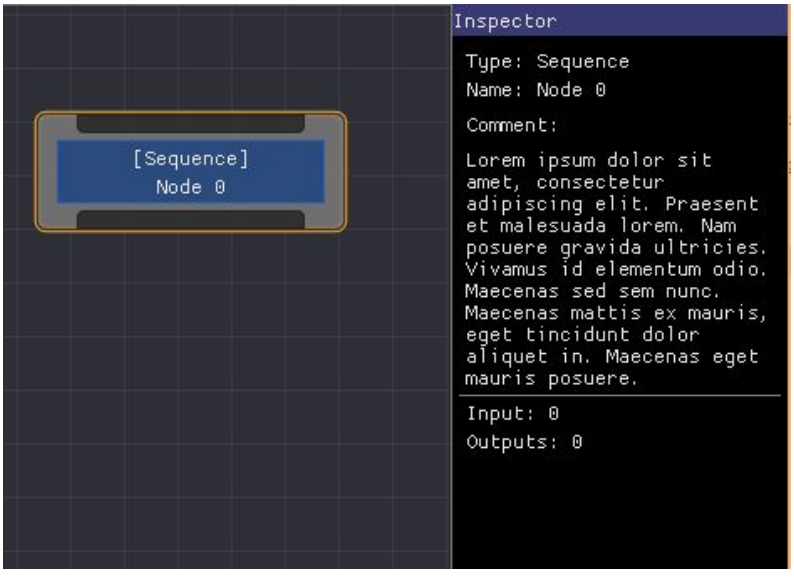
5.3 BeeT 0.01 Node creation



5.4 BeeT 0.01 Multi-selection

In the inspector window, when only one node is selected, it appears the node's information.

For now, it only shows some basic testing information like:



- **Type**
- **Name**
- **Comment**: to write information about the node
- Number of **inputs** connected
- Number of **outputs** connected

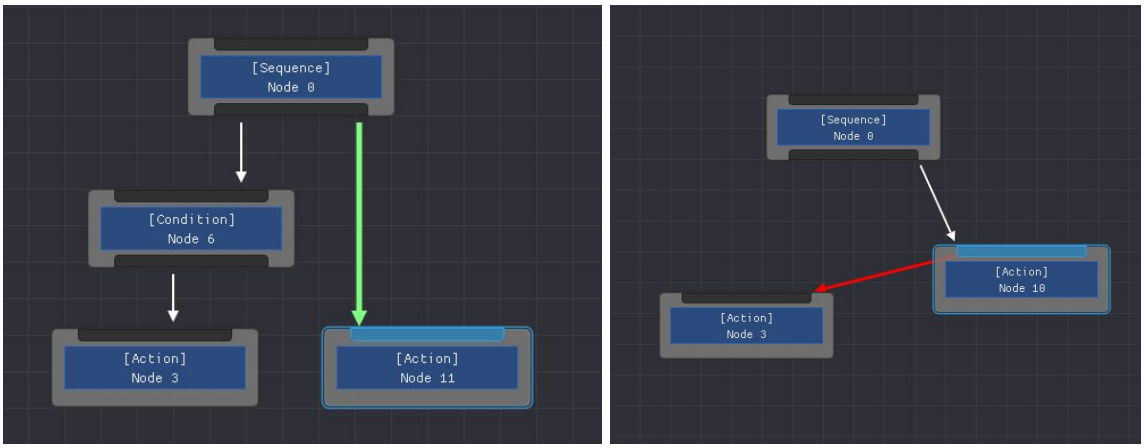
5.5 BeeT 0.01 Inspector

Dragging a node's input or output pin with the left mouse button to another node creates a link between them. This way, we can create the hierarchy of the tree.

The links can only be created successfully following the basic rules of behavior trees:

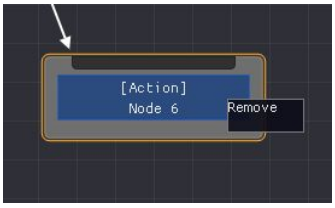
- A node can only have one parent node
- All nodes must be connected to the root hierarchy to be executed
- Recursion is not allowed
- A node output cannot be connected to its own input

To help visualize these rules the link is highlighted in green when the connection is possible and with red when the connection is not allowed.

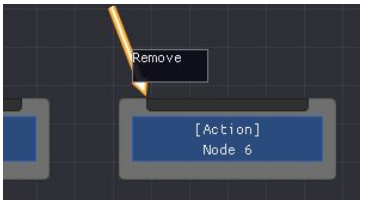


5.6 BeeT 0.01 linking nodes

Finally, nodes and links can be removed with a left mouse click above them.



5.7 BeeT 0.01 remove node



5.8 BeeT 0.01 remove link

5.2 Vertical Slice

Objectives:
- A visual node editor to: create, link and delete nodes
- A serialization system to save and load behavior trees
- Library initialization and cleanup functions
- Library function to load behavior trees generated by the editor
- Create the behavior tree structure while loading the file

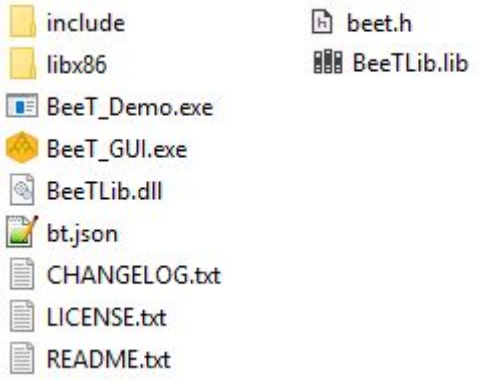
At the beginning of the Vertical Slice we decided to modify a little bit our current methodology. Now that the real development had begun and the exploration phase was over, we need it a solid methodology to keep track of our progress. The result was a series of weekly builds that showed the progress of the project.

A total of three builds were released during this stage. We will analyze each one separately to take detailed look of the changes.

5.2.1 BeeT 0.02

In this version the library was already started, for this reason the files that the release includes differ from the previous ones. From that point, all the future versions follow the same pattern:

- A *include* folder with the header of the library inside
- A *libx86* folder with the '.lib' of the library
- An executable named *BeeTDemo* to test the library functionality
- An executable named *BeeTGUI* with the visual editor to create trees
- A '.dll' file of the library
- A '.json' file with a behavior tree example to be used by the demo
- A *LICENSE* file with the license of the software
- A *README* file with an explanation of the software
- A *CHANGELOG* file with all the changes made from one version to another



5.9 BeeT 0.02 release content

The first thing to complete was the creation of the library project with the most basic functions: init and close. Next, a function to load a behavior tree either from a file or from memory. To test the library, another project was created named Demo. The Demo contains a series of tests to simulate the use of the library and assure every modification works as expected.

While doing the loading of the behavior tree file, the need of implementing a library to handle all the filesystem arised. The chosen one was PhysFS, mainly because we already worked with it in the past and covered our needs.

Finally, the header file of our library looked like this:

```
void Init();
void Shutdown();

int LoadBehaviorTree(const char* buffer, int size);

/*
*     Loads a BehaviorTree from a file path.
*     \param filename The file path
*     \return The uid of the Behavior Tree on SUCCESS, -1 on FAIL.
*/
int LoadBehaviorTree(const char* filename);
// Get the number of Behavior Trees loaded in memory.
size_t BehaviorTreeCount();
```

5.10 BeeT 0.02 library header

On the other part, the editor, a couple of improvements were made:

- Allow window resize
- User cannot longer link nodes in a loop

Furthermore, a dialog window was added to select the file to open or save. This allowed the user to create more than one behavior tree and improved the user experience.

- Save/Load directory
- Files inside that directory
- Input box: to write the file name
- Accept/Cancel options

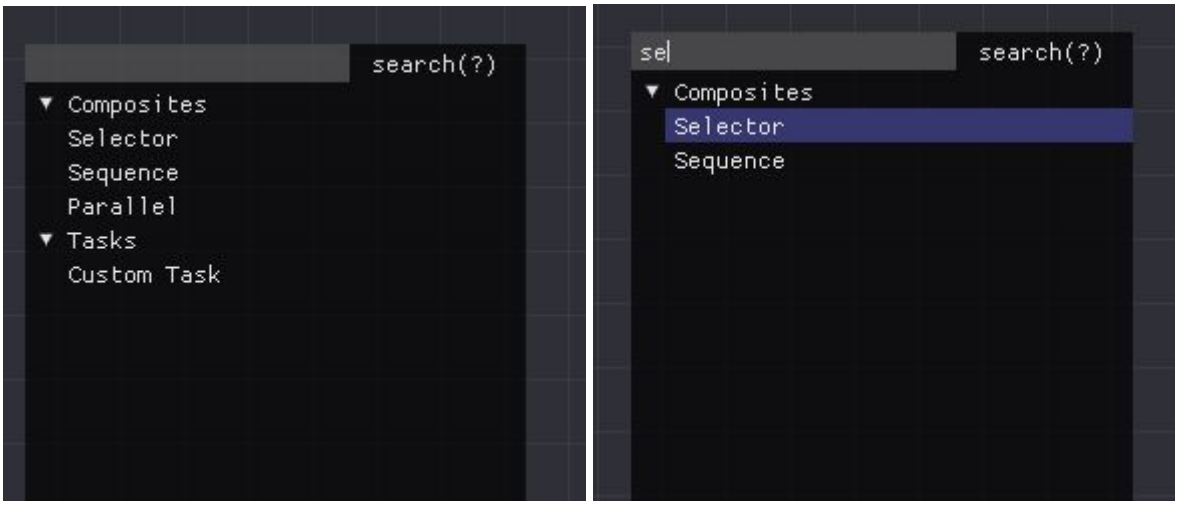
5.11 BeeT 0.02 save/load window

5.2.2 BeeT 0.03

In this version a major change was made in the library: we switched from C++ to C. The reason behind it was that our director recommended this change to facilitate the integration of the library.

On the editor side, we added new functionality and did some minor modifications to improve the tool's usability.

The node's creation process became more agile with the new window:

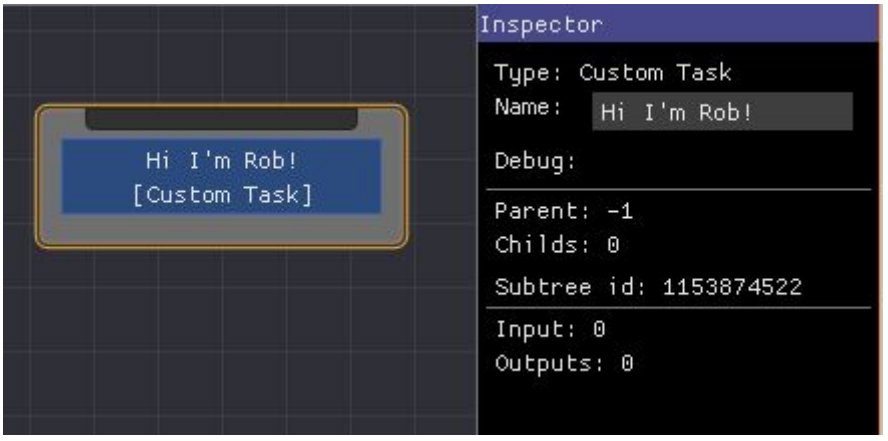


5.12 BeeT 0.03 create node window

Features:

- Filter search: to find nodes by name or category
- Keyboard auto-focus on filter search
- Enter key as a shortcut for confirmation
- Nodes grouped by category

Additionally, node's name can be modified in the *Inspector* window and be shown inside the node. After selecting the node type when creating the keyboard focus is set to the node's name box to quickly rename the new node.



5.13 BeeT 0.03 Inspector rename node

Behavior tree files were changed to save it with the '.json' extension instead of the previous '.txt' for consistency, considering that we already saved the file in json format.

Finally, some minor changes were made to improve the quality of the tool:

- Node's comment option was removed: we found out it was a useless feature.
- The root node of the tree is not selectable. All trees must have a root node and we do not want the user to modify it.
- The canvas node editor is centered with the root node at the start
- Automatically selects the new node after creating
- Quit option was moved from *Menu Bar* to *File* for consistency with other softwares

5.2.3 BeeT 0.1

BeeT 0.1 is the final version of the Vertical Slice stage. The changes made from the previous version were most of them internally: code structure, optimization, error solving, etc.

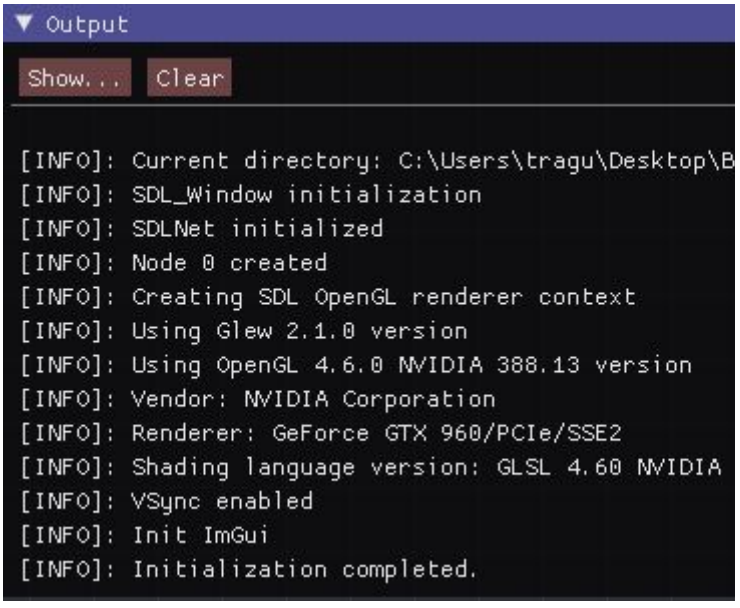
The *Demo* project was restructured to have the code organized in 'tests' that only targets one specific section to evaluate. This way, we can analyze the performance of each new feature we add.

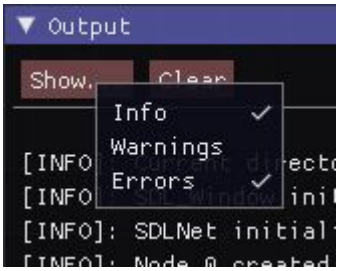
The *Library* project was also restructured to a more solid and robust code following the architecture of SDL as an example. In addition, we solved all warnings that appeared when compiling to prevent future issues.

Finally, in the *Editor* project we added a new window to display all the messages, warnings and errors that the application throws when running. The messages can be filtered by three different categories:

1. INFO: for informative messages
2. WARNING: for warnings that should be avoided
3. ERROR: for things that went wrong

There is also an option to clear all messages that helps finding the specific message that the user is looking for. This tool was not a task of this stage but we found out that it was really necessary at that point of the development process.





5.15 BeeT 0.1 output filters

5.14 BeeT 0.1 output window

5.3 Alpha

Objectives:
- A blackboard in the editor to create and edit variables
- Save and load the blackboard with the behavior tree save file
- Use the variables of the blackboard in the library
- The library is able to run behavior trees created by the editor

5.3.1 BeeT 0.11

Once the library was restructured and loaded the behavior tree file generated with the *Editor* we could start implementing the functionality of each node as well as the behavior tree logic.

As we said early, we will implement the event driven approach to handle the tree behavior. We will explain this implementation in more detail to understand the basics of how it works.

The general idea is that each node has an observer that will be notified once the node is completed. Node's active behaviors are pushed on to the tree and this only executes the active ones. This way the tree becomes similar to an interpreter as it passes events between nodes.

A pseudo-code of this implementation could be:

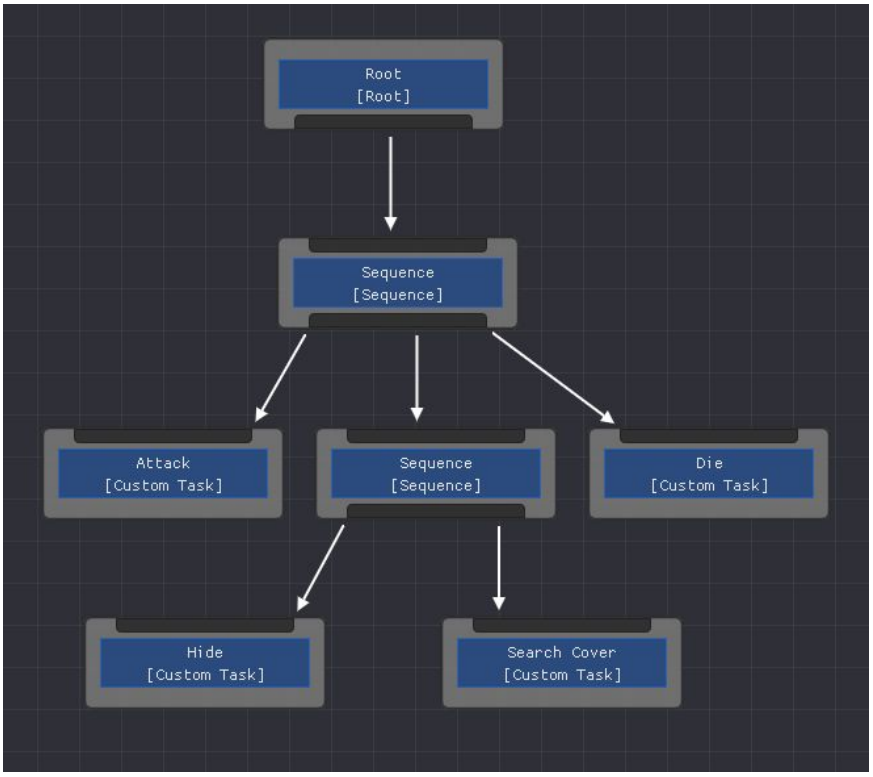
```
1. bt (BehaviorTree)
2. bt.push_back_behavior(rootNode)
3. bt.Update() -> bt.behaviorQueue.front.Update() "Node Update"
```

```
"Node Update":
    1. If has childs -> bt.push_back_behavior(nodeChild)
    2. Do stuff
    3. return status
        a. FAILED or SUCCEED depending on internal logic
        b. RUNNING if waiting for child response
```

```
4. bt receives the node return status. If:
    a. RUNNING: push node back to Queue
    b. FAILED or SUCCEED: bt calls node observer -> "OnNodeComplete"
```

```
"OnNodeComplete":
    1. Own function for each node type.
       For this example assume node = Sequence.
    2. If complete status is:
        a. SUCCEED: push next child to b.push_back_behavior(child).
           if next child is NULL change status from RUNNING to SUCCEED
        b. FAILED: change status from RUNNING to FAILED.
```

Once this was implemented in the library we created a new test for the *Demo*. Consists of a simple behavior tree like this:

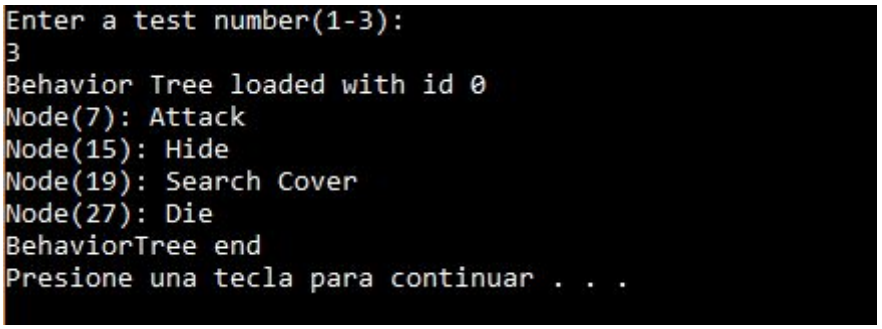


5.16 BeeT 0.11 Demo Test 3

The internal process of the tree should be:

1. *Root* node runs its child *Sequence*
2. *Sequence* runs all its childs starting from *Attack* until all are completed or one returns FAILURE.
3. *Attack* is a custom task that:
    a. Prints the node's name
    b. Returns SUCCESS
4. All *Custom Task* have the same behavior so the execution order is:
    a. Root
    b. Sequence
    c. Attack
    d. Sequence
    e. Hide
    f. Search cover
    g. Die
    h. At this point the tree ends

If we run this test (test nº 3), the output is:



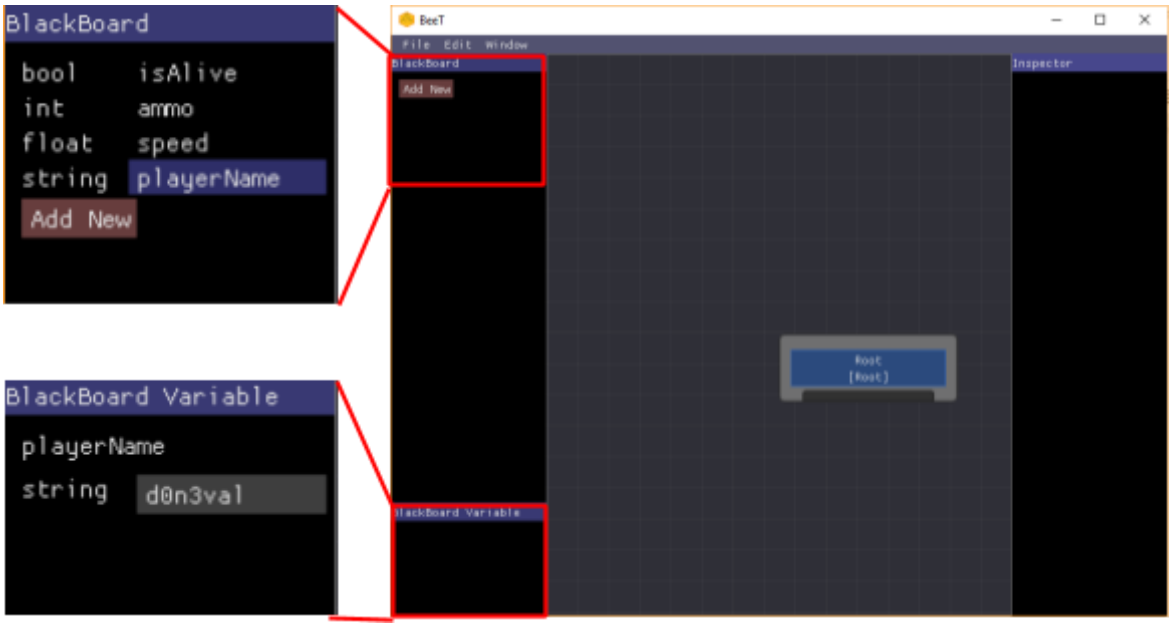
5.17 BeeT 0.11 Demo Test 3 output

As we can see, the *Custom Task* nodes print their name in the correct order, confirming that our tree implementation works as expected.

5.3.2 BeeT 0.2

This is the final release for the Alpha stage. In this version we focused on implementing the blackboard.

We started by the *Editor* since it is the one that will create the variables of the blackboard and will serialize them in the behavior tree file. A new window was added to display the blackboard variables and to create them. The user can specify the variable type and name. Note that the name must be unique to identify each variable. If the user has not set a name, the program automatically sets a unique name for the variable.



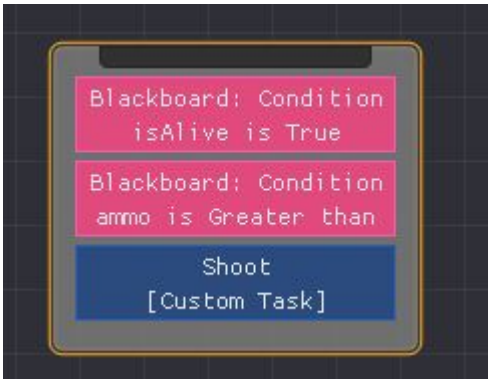
5.18 BeeT 0.2 editor blackboard

At the left bottom there is a window to edit the value of the selected variable.

Once the edition part was done, the next step was to serialize all this information in the behavior tree file. Both, the editor and the library, had to be able to read this file and load all the relevant data.

To end, the last step was to add decorators inside each node. A decorator is like the conditional node of the common behavior trees. It is a conditional that checks the value of a variable of the blackboard and returns true or false depending on the condition. Decorators are executed before the node, if all the conditions (decorators) are passed the node is executed, if not, the node returns failure.

Here is an example:



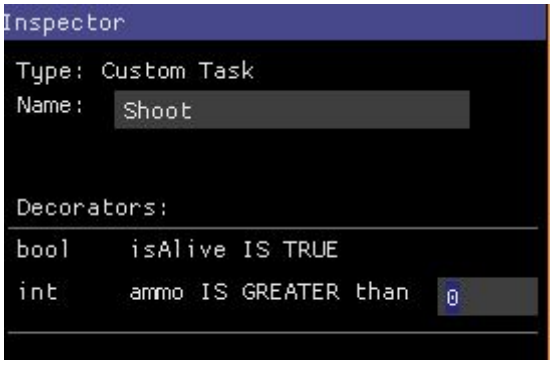
The task *Shoot* will be executed only when:

isAlive is true AND ammo is greater than 0

When one or both of these conditions are not meet the entity will stop shooting.

5.19 BeeT 0.2 node with decorators

When a node is selected, the user can set the condition of each decorator in the *Inspector* window:



5.20 BeeT 0.2 Inspector node decorators

For each type of variable there are different kind of conditionals. Below there is a list of each variable type with its corresponding group of conditionals:

- Bool:
  - is true
  - is false
- Int/Float:
  - is equal to <number>
  - is not equal to <number>
  - is greater than <number>
  - is less than <number>
  - is greater or equal to <number>
  - is less or equal to <number>
  -

- String:
    - is equal to <string>
    - is not equal to <string>
    - contains <string>
    - not contains <string>

Finally, the last step was to load all this information in the *Library* and run the decorator checks before each node. To test all the features we created two more tests: test4 and test5.

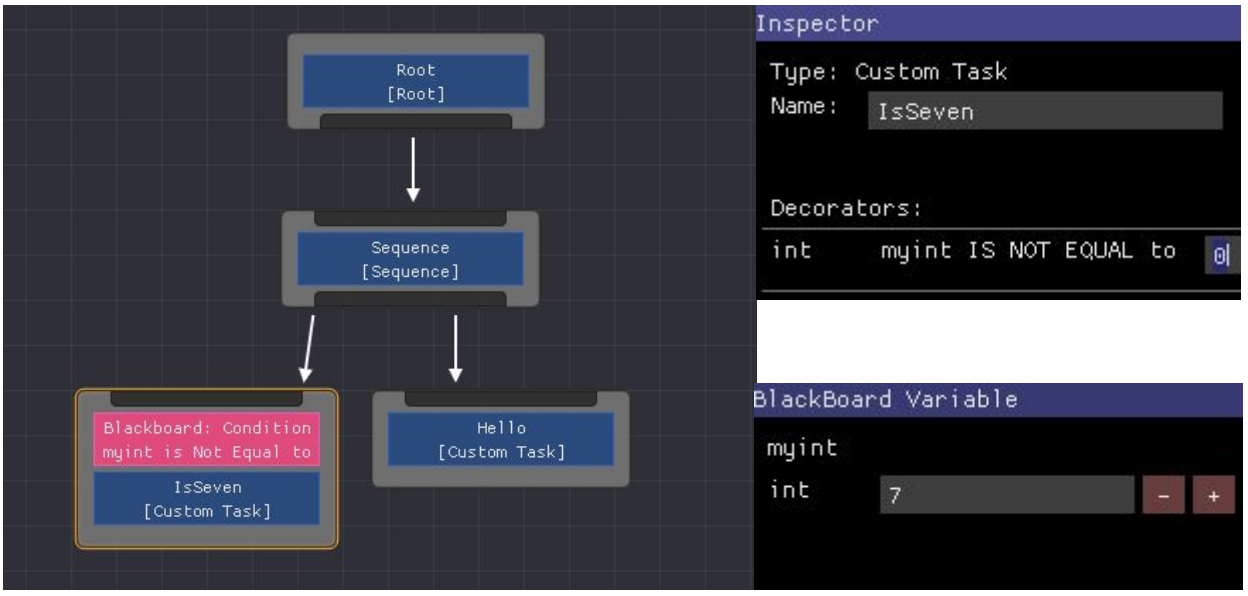
Test4: the objective of this test was to try the functions to set and get a variable of the blackboard from the *Library*. The test prints the variable value at the beginning and then sets a new value. Internally what we do is a get, set and another get. The result can be seen below:

```
Enter a test number(1-5):
4
Behavior Tree loaded with id 0
mybool = 1
myint = 7
myfloat = 6.555500
mystring = Hello World
---------------------------------
mybool = 0
myint = 4
myfloat = 123.456001
mystring = Goodbye World
```

5.21 BeeT 0.2 Test 4 output

Test5: in this test we run a behavior tree that contains a decorator in one of its nodes. It prints the node's name if it is a task. With that, we can know if the tree has reached one node or not, if it prints it or not.

The behavior tree is the following:



5.22 BeeT 0.2 Test 5 'blackboard.json' file in Editor

Accordingly, the steps by order will be:

Root - Sequence - Check decorator conditional:

a- Pass: print IsSeven - print Hello

b- Not pass: end of execution

And the decorator check:

- Decorator will check if blackboard variable myint is not equal to 0.
- The value of myint is 7.
- The condition will be: is 7(myint) not equal to 0?
- Answer: yes

Therefore the decorator will pass and the path that the tree will take will be 'a'.

If we run this test the output is the following:

```
Running Behavior Tree:
----------------------------------------
IsSeven
Hello
BehaviorTree end
```

5.23 BeeT 0.2 Test 5 output

We can also open the 'blackboard.json' file in the *Editor* and add more decorators or change the condition so that it does not pass. If we do so, the output will not print any of the two task names because it will never execute any of them.

5.4 Beta

Objectives:
- Connect the library and the editor to debug the game
- Visualize the behavior trees running in the game in the debugger
- Create a set of tools to debug behavior trees

5.4.1 BeeT 0.5

The goal of this release was to implement the debugger, but in order to do that, first we had to find a way to communicate between the library and the editor application. It was the time to start building the network system to connect both applications.

As we mentioned at the beginning we would use SDLNet_2.0 to let it handle all the sockets functionality. However, SDLNet only provides the connection between sockets, we have to implement the structure to start, receive and update the connections as well as building the structure to encapsulate all the information into a packet to send it and receive it.

The structure we followed was:

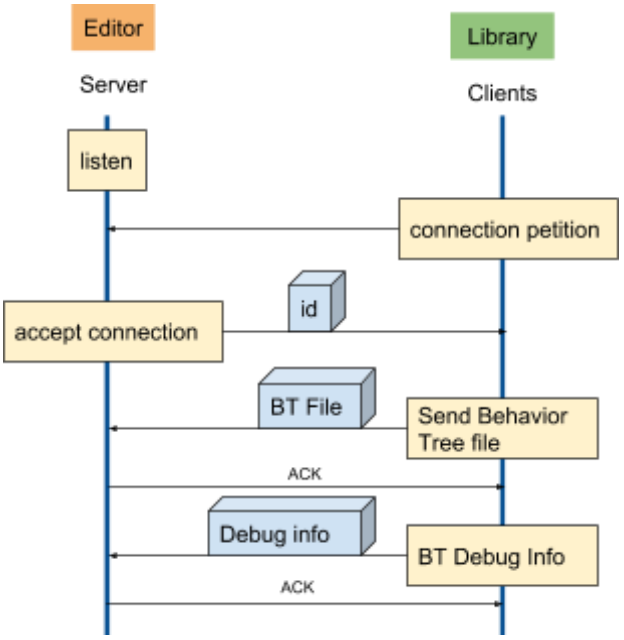


Figure 5.24 Editor-library connection diagram

The editor acts like a server and every behavior tree that needs to be debugged is sent by the library as a new client. First we do the typical connection request, acknowledgment, sent a unique identifier, etc. Once the connection is successfully established we can proceed to send data.

There are two types of data that we send from the library to the editor, in the following order:

1- Behavior Tree file: is the file from which the library has loaded the BT. It contains all the information about the BT. With that information, the editor can recreate the tree at its initial state.

2- Debug info: is all the data collected from the library from that specific BT that we are debugging. It contains all the changes, if there is any, made to the BT at a given frame.

To each specific change we have call it sample and can contain just one concrete modification to the tree:

- The current node has changed to another node
- A node returned with a different state than the previous frame
- A decorator (condition) has passed or not
- The value of a blackboard variable has changed
- The behavior tree has ended and the root node has returned with a final state

At the end of each frame all behavior trees with the debug flag activated are asked to collect all the samples generated that frame to sent it in a package to the editor afterwards. In order to display the samples correctly in the editor, a timestamp is saved at the creation of each sample to make sure the editor receives the samples in the correct order.

This was the first part of the debug process: collect and send data. The next step was to receive that data and interpret it in the editor. A new mode was added, the Debug mode, that changed all the appearance of the editor to show only the debug tools.

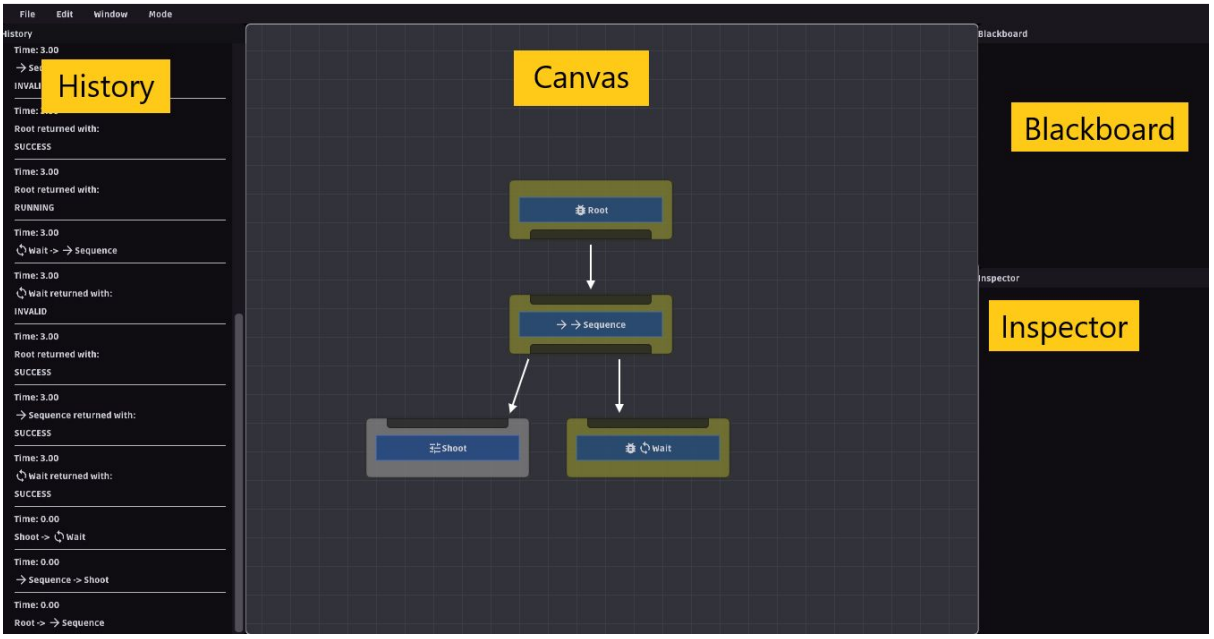


Figure 5.25 Debugger windows

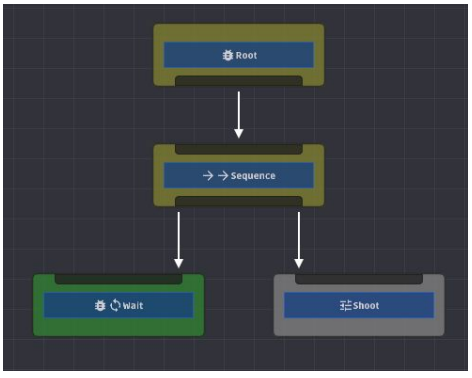
At the left we have the history window where it shows the record of all the changes that the behavior tree has been through. Clicking one of them takes the state of the tree to that specific moment in time. With this, it is easy to track the development of the tree over time.

A history element shows information about:

- Time since the BT started running
- Type of sample (in this case a node returned something different than the previous frame)
- Additional data specific to each sample type (in this case the state returned by the node)

Figure 5.26 Debugger history samples

In the canvas section, there is the behavior tree with the nodes colored depending on its state.



- Yellow means the node is running (executing)
- Green means the node returned with success.
- Red the same as before but returned fail
- Grey is the node hasn't been executed yet.

Figure 5.27 Tree color code in debug

To test the first iteration of the debug tool we made a simple game with one enemy that ran a behavior tree meanwhile we were debugging it with the editor. The behavior is the one of the left and at the right there is a screenshot of the game. The player controls the bee and the flower is the enemy. This behavior, what it does is, wait for 3 seconds and shoot (and repeat infinitely).

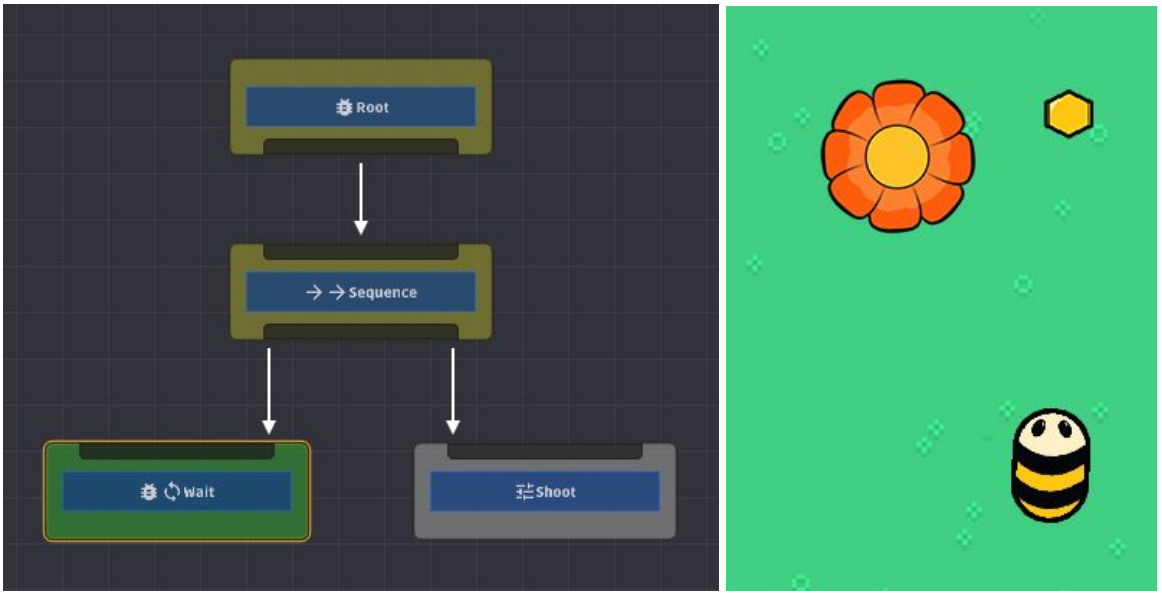


Figure 5.28 Comparison of a behavior tree with its application in a game

Although the objectives of this stage were completed, we realized that it needed more iterations to refine the debug tools until it is something worth using. We already had the tool that we designed at the beginning but in a real scenario we found out what was missing and what was unnecessary. That was the purpose of the final stage, to transform the tool into something useful.

5.5 Gold

Objectives:
- Test the tool to detect functionality problems or user experience improvement points
- Polish
- QA and bug fixing

For the Gold stage we focused on finding out the weak points of the library to fix them. To do so, we thought that the best way was to use the library in a real game. We had already started doing a mini-game in the Beta but this time the game needed to be more complex

to use the whole potential of the library. We are not going to explain the game here because it is not related with the main topic, it was only used as a tool to help us detecting the potential problems.

From now on, we are going to describe each improvement briefly to have a general vision of the evolution of the library and the tool from this point.

5.5.1 BeeT 0.51

When a node that allows multiple children is selected (sequence and selector), in the inspector appears an option to choose the execution order of the childs. Usually, the childs are placed from left to right in order of execution, but considering that the position of the nodes is something the user decides we couldn't know the order which the user wants to execute every child. For this reason, we allow him/her to set the order in the inspector independently of the node position.

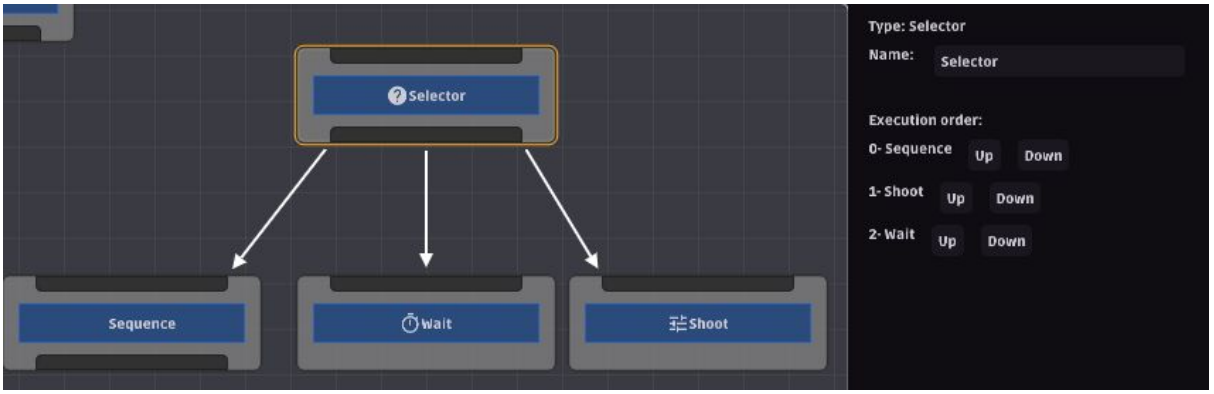


Figure 5.29 Setting order of child nodes

In the image above we could think the order should be: Sequence, Wait and Shoot. However the user decided to run Shoot before Wait as it has been set in the Inspector.

The task Wait has been added by default in the editor since it is a common node that is used very often. It acts the same as a task node that its functionality is to pause the execution for a period of time.

Blackboard variables must have different names because they are used as identifiers when setting or getting the value from the library. To avoid errors, we forbid the user to create two variables with the same name, a new name is generated automatically if this happens.

There has been a visual improvement in the debug window to facilitate what is happening during debugging. The current node that is running is highlighted in red and the arrows that connect two nodes are painted with the state of the connection: running, fail, success or idle. There is also an animation for the links which are in the running state to quickly spot the path of execution of the tree

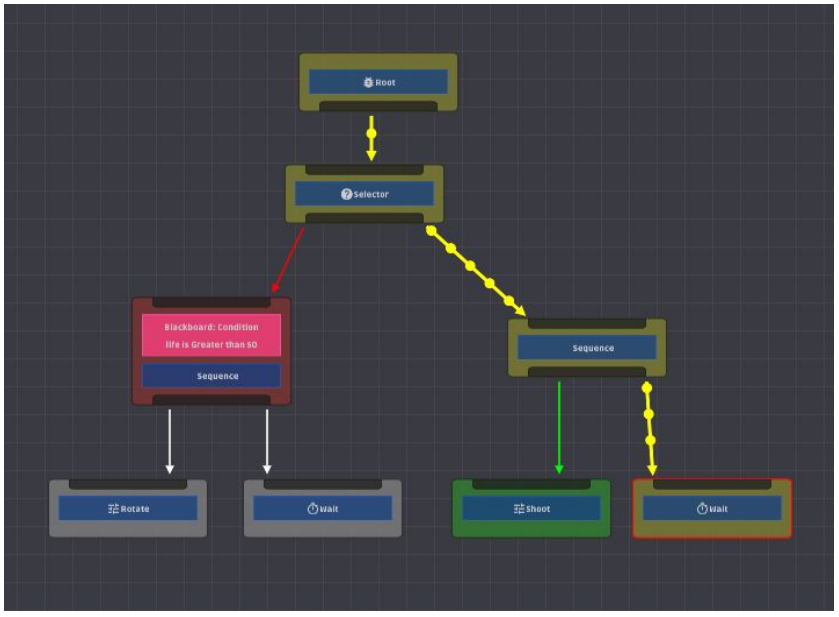


Figure 5.30 Debug screenshot

5.5.2 BeeT 0.6

This was the second iteration of the final product. In this release we extended a little bit the library functionality with:

- The callback function to execute the Task nodes now have an extra parameter to indicate the function of the task: OnInit, Update, OnFinish.

```
NodeStatus beetCallbackFunc (unsigned int btId, const char* taskId, NodeFunction func);
```

    - OnInit: called the first time the Task node is executed
    - Update: called every tick
    - OnFinish: called when the Task is no longer running and returns a final state
- The root node restarts itself once the tree is completed. This way, the tree is always running unless the user stops it.
- The function to init the debugger now has an extra parameter: IP. Allows the user to set not only the port, but also the IP to make the connection between library and editor.

- The decorator condition have the option to be checked only once, the first time the node starts running, or each tick. An example: we want to check if we have enough ammo only once and start reloading (decorator checked only at the start), or we want to check every time if the player sees us while we are reloading (decorator checked each tick) to stop reloading and search cover.

Aside of the extended functionality we also added a new type of node: parallel. As we explained in a previous chapter, the parallel node allows to run multiple nodes at the same time. However, to take advantage of the event driven approach we have implemented we used the same system UE4 uses in their parallel node.

Parallel node only allows two children: one main node that must be a task and the second node can be a complete subtree. The subtree repeats (if finishes) while the main node is running. When this happens, the parallel node returns the state of the main node and the whole subtree stops running.

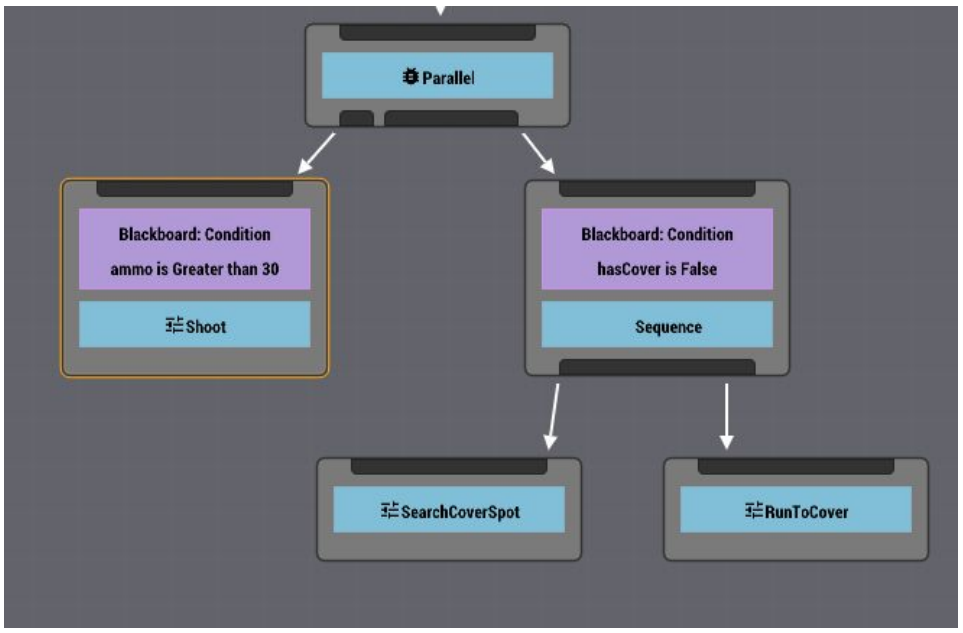


Figure 5.31 Parallel node

In this example the main node is Shoot while the others are part of a subtree. What it does is:

- Shooting while ammo is greater than 30. If it is less than 30 the parallel node stops running
- While this happens it is constantly checking if has cover, if not, searches for one and goes there

The result is a entity that is shooting all the time and tries to stay under cover while doing so.

Finally, regarding the library improvements we also:

- Optimized the number of running nodes. Nodes that are waiting a child response do not tick (ex: selector, sequence)
- Cleaned up some memory leaks
- Reduced the number of samples sended while debugging by avoiding sending repeat samples of the previous frame

For the other part, the Editor, the improvements we did for this version were:

- A new way to show the evolution of the behavior tree while debugging. Like a video, where you travel along the samples:

Figure 5.32 Timeline evolution of a tree

- An option to search a node by ID:

Once the ID is set the canvas focuses the node selected with the same ID.

Figure 5.33 Find node option

- Option to show the nodes ID under the Show menu. Displays the node's ID in the canvas.
- Multiple tabs while debugging more than one behavior tree
- Parallel node. Explained above in the library improvements section.

- Option to set the IP and port for the debug connection as well as an activation button.

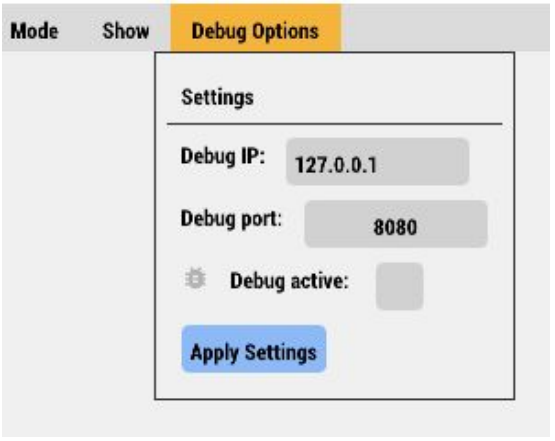


Figure 5.34 Debug network options

- Decorator option in the Inspector to tick every frame or only at startup. Also a visible change of color to distinguish decorators by their check type:

Figure 5.35 Decorator tick options

It is worth mentioning that the whole user interface of the Editor was modified to have a different visual aspect. However, we are not going to enter into detail because it was part of an experimentation to find the visual aspect that fit with the tool. It may be subject to future changes.

5.5.3 BeeT 1.0

In this release we only added a few functions and we prepared the library for the official gold release.

The functions added are:

- Pause and resume a behavior tree: as the name indicates it pauses/resumes the execution of a specific behavior tree
- Close a behavior tree and remove it: to indicate to the library that the behavior tree is no longer in use and can be removed safely from memory

5.6 Performance

We have implemented the following behavior tree in a demo as the behavior of the enemy:

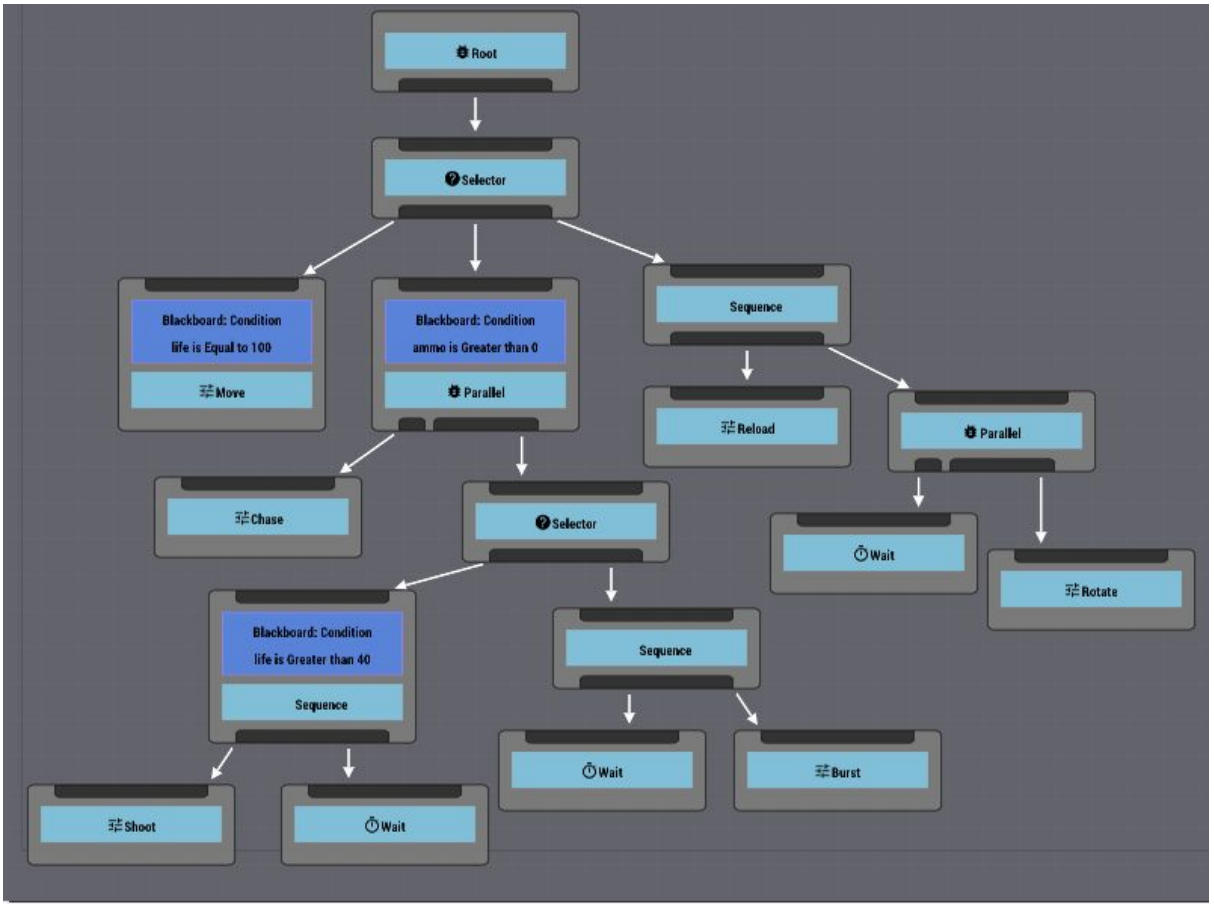


Figure 5.36 Behavior tree used in the performance test

It includes all types of nodes as well as some decorators linked with variables of the blackboard. We are not going to explain what this tree does as it is not relevant to understand it to test the performance.

To make the performance test we have calculated the time that the function Tick of the library takes every frame. However, we have subtracted the time it takes to perform each node Task because we do not have control over what these tasks do, it is something the user controls. We created copies of the enemy mentioned above to have multiple trees running at the same time. For the tests we collected 500 samples (500 ticks) and calculated the average time, in milliseconds, it took to update the library. The results are:

Num of BTs	Avg. time (ms)	Min time (ms)	Max time (ms)
20	0.05	0.02	0.33
200	0.45	0.25	0.93
2,000	2.26	0.30	3.79

Table 5.1 Library performance impact in ms

The number of trees running will be different for every game as well as the complexity of the tree. Nonetheless, we can make a broad approximation of the number of trees running and the complexity of all of them and take as a reference the timings we took of 200 behavior trees running simultaneously: 0.45ms. This proves that our library does not affect in a high degree at the performance of the game. To have a better vision of the impact of our library in a game we compared the time it took to update the library with the total time a game have to update its frame at 60 FPS (16.66 ms) and at 30 FPS (33.33 ms).

Num of BTs	Avg. time (ms)	60 FPS	30 FPS
20	0.05	0.31%	0.15%
200	0.45	2.71%	1.36%
2,000	2.26	13.59%	6.79%

Table 5.2 Library performance impact for a 60 FPS and 30 FPS game

These tests have been executed in a laptop with: CPU (Intel Core i7-4500U 1.80GHz 2.40GHz, RAM (8 GB), GPU (Nvidia GeForce GT 740M), system (Windows 10 x64).

6. Conclusions and future work

At the beginning of this project and after a little bit of research, we found out there was nothing like what we aimed to do. There were some similarities with professional tools already in the market so we used them as reference to built our tool.

Evaluating the objectives individually we can affirm that we did reach our initial objectives. However, during the development process we encountered new necessities that we did not plan at the beginning. Some of the improvements that we could do in the future could be: increase the number of built-in nodes in the editor, modify variables of the blackboard directly from the tree, a easier interface to debug large number of behavior trees at the same time (for open world games with a large number of NPCs for example), increase the type of variables available in the blackboard (add: arrays, vector2, vector3, ….).

As we could appreciate in the previous chapter, the performance is excellent as there is a negligible impact making it viable for a practical use. Furthermore, it has a simple integration due to the few calls it has to initialize and use the library. As a result the user does not have to worry about the decision making because the library has masked that functionality and has made visible only the most basic.

The editor we made can be considered capable enough to create a broad number of behavior trees for different kind of games, not too complex, but still interesting. Combined with the editor there is the debugger which facilitates the task of fixing the tree to accomplish the desired behavior.

There is still work to do, mainly in the editor application. As we increase the number of options we offer to the user in the editor, the library will also need to expand to provide the same functionality. It is hard to define or list what needs next as it is very project dependant. Every project will have different needs and will expand the library in a different direction. It is necessary to point out that one of the first objectives was to have a solid base where each developer could take the library in their desired direction. With all that, we could say the best option is to let people use the library and collect all the feedback from their projects. The future work will be determined by the own users.

To conclude, we present the latest build of our project that can be found at this link.

7. Bibliography

Recast navigation [Online]. GitHub repository, URL <https://github.com/recastnavigation/recastnavigation>. [Consult February 10th 2018].

2nd Behavior Trees generation [Online]. Video, URL <https://www.youtube.com/watch?v=n4aREFb3SsU>. [Consult February 10th 2018].

Introduction to AI - Part 2: UE4 Behavior Tree Specifics [Online]. Video, URL <https://www.youtube.com/watch?v=Bvs8t6eRlqc>. [Consult February 10th 2018].

Are Behavior Trees a Thing of the Past [Online]. Blog, URL <https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php>. [Consult February 12th 2018].

GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI [Online]. Blog, URL <https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php>. [Consult February 12th 2018].

6 Top Game Engines in 2017 [Online]. Blog, URL <http://www.discoversdk.com/blog/6-top-game-engines-in-2017>. [Consult February 18th 2018].

How UE4 Behavior Trees Differ [Online]. Web page, URL <https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html>. [Consult February 22nd 2018].

Behavior Designer [Online]. Online Store, URL <https://assetstore.unity.com/packages/tools/visual-scripting/behavior-designer-behavior-trees-for-everyone-15277>. [Consult February 22nd 2018].

NodeCanvas [Online]. Online Store, URL <https://assetstore.unity.com/packages/tools/visual-scripting/nodecanvas-14914>. [Consult February 22nd 2018].

NPBehave [Online].Online Store , URL <https://assetstore.unity.com/packages/tools/ai/npbehave-event-driven-code-based-behaviour-trees-75884>. [Consult February 22nd 2018].

3 Differences Between Scrum and Kanban You Need to Know [Online]. Blog, URL <https://www.cprime.com/2015/02/3-differences-between-scrum-and-kanban-you-need-to-know>. [Consult February 24th 2018].

Artificial Intelligence [Online]. Web page, URL <https://en.wikipedia.org/wiki/Artificial_intelligence>. [Consult February 25th 2018].

Finite-State machine [Online]. Web page, URL <https://en.wikipedia.org/wiki/Finite-state_machine>. [Consult February 25th 2018].

About GDC [Online]. , URL <http://www.gdconf.com/aboutgdc>. [Consult February 25th 2018].

Modular Behavior Tree [Online]. Web page, URL <http://docs.cryengine.com/display/CEPROG/Modular+Behavior+Tree>. [Consult February 25th 2018].

GameAIPro Chapter 06: The Behavior Tree Starter Kit[Online]. Portable Document, URL <http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf>. [Consult February 25th 2018].

8. Annexes

[1]Behavior tree.

A behavior tree is a tool to emulate behavior using a tree of actions. This technique is widely used in video game artificial intelligence to create the behavior of enemies as well as other entities.

The way behavior trees work is by traversing all the tree from the root node to a child that returns something, it is usually a task/action. This is done every tick or update and creates the behavior of the entity. As we mentioned before each node returns 'something', that can be:

- Success: if the node has finished its task successfully
- Failure: if the node failed the task.
- Running: if the node is still performing the task.

For more information about behavior trees read this post.

[2]Second generation of behavior trees.

During the evolution of the design patterns of behavior trees over the years there have been a differentiation between the first implementations and the new ones. This second generation is defined by:

- Larger and deeper trees with more nodes
- Smaller powerful nodes that combine better together
- Data is shared between multiple behavior trees instances
- Heavily optimized to improve scalability
- Written as a reusable library that can be applied to any game logic

There are two main design patterns in this new generation:

- Data oriented: memory is allocated at initialization. Nodes come in memory after their parent reducing the number of cache misses and reducing memory consumption by accessing each node by indexes and offsets without the need to rely on pointers.

- Event-driven: nodes' return states are sent through events managed by the tree. There is no need to traverse the whole tree every tick, instead the starting point is set to the last updated node. This guarantees a minimal computation